An Immutability Type System for Classes and Objects: Improvements, Experiments, and Comparisons

by

Lian Sun

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Lian Sun 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Lian Sun is the sole author for Chapter 1, 3 and 5. They are written under the supervision of Professor Werner Dietl, and are not yet published.

The introduction of PICO in Section 2.4, and the whole program inference experiment in Section 4.4 contains previous work by Mier Ta and Professor Werner Dietl. The remaining sections of Chapters 2 and 4 are solely completed by Lian Sun under the supervision of Professor Werner Dietl, and are not yet published.

Abstract

Mutability, the ability for an object to change, is frequently cited as one of the sources of software problems. Ensuring the immutability of objects opens opportunities for optimizations, e.g., removing the need for locks in a concurrent environment for an immutable object. This thesis explores an approach to analyze immutability of classes and objects by using static analysis with pluggable type systems. A properly implemented pluggable type system can statically analyze the mutability property of an object without execution. This thesis presents (1) the analysis of some previous work, including Javari, ReIm, and Glacier, (2) improvements to a pluggable type system, PICO, to enhance the soundness of the formalization and to improve the user experience, and (3) experiments with the enhanced PICO with real projects, and comparisons with the results of the previous work.

PICO is an immutability type system that analyzes and enforces the mutability property of an object so that a mutation on an immutable object can be statically detected. Although many modern programming languages have various means of declaring this property, PICO provides an easier, more flexible, and foolproof way to declare the mutability property of a class by automating the check of immutability.

While PICO is a novel work in improving the flexibility of the immutability type system, it has certain bad designs for defaulting in parts of the immutability rules. Such bad designs would lead to the risk of allowing the mutation of an immutable object, known as the false negative. To solve this problem, this thesis provides more sound formalization to fix the false negative.

Also, PICO contains counterintuitive logic, such as unsafe defaulting. To solve the counterintuitive logic, this thesis presents a new defaulting scheme for PICO, and reports various minor changes made to improve the user-friendliness during the type checking process.

This thesis conducts experiments on small code snippets and large real-world projects, and also compares the new PICO with previous works on immutability to find more potential problems and demonstrates the flexibility and usability of PICO compared with previous projects, e.g., Glacier.

Acknowledgements

I would like to give my most sincere thanks to my supervisor Professor Werner Dietl, for his constant support and guidance. I would like to thank my readers, Professor Arie Gurfinkel and Professor Mahesh Tripunitara, for their time and feedback. I also would like to thank all my teammates in the research group, especially Tongtong Xiang, Weitian Xing, and Di Wang, for their help and bringing joy to the two years of my study. I am also thankful to a former teammate, Mier Ta, for his brilliant and insightful previous work.

Dedication

This is dedicated to my parents who have always wanted the best for me.

Table of Contents

List of Figures **x**

List of Tables **xi**

1 Introduction **1**

2 Background and Related Work **5**

2.1 Pluggable Type Systems 5

2.2 Checker Framework and Inference 7

2.2.1 Checker Framework 7

2.2.2 Checker Framework Inference 9

2.3 Mutability Type Systems 11

2.3.1 Javari 12

2.3.2 ReIm and ReImInfer 13

2.3.3 Glacier 14

2.4 PICO: Practical Immutability for Class and Objects 16

3 Improvements to the PICO Type System **22**

3.1 Class Bound Use 22

3.1.1 Definitions of Different Bounds 22

3.1.2 Class Bound Handling 23

3.1.3 Anonymous Class Bound in Inference 24
3.2 Polymorphic Substitution and Type Variable Resolution 25
3.2.1 Introduction to Polymorphic Qualifiers 25
3.2.2 Polymorphic Substitution and Type Argument Resolution 26
3.2.3 Polymorphic Substitution and Viewpoint adaptation 28
3.3 Viewpoint Adapted Subtype 30
3.3.1 The Issue of Formalization 30
3.3.2 Definition of Viewpoint Adapted Subtype 31
3.3.3 Updated Formalization 32
3.4 Mutability of Enums 34
3.4.1 The Enum Type of Java 34
3.4.2 Revised Mutability Constraints for Enums 35
3.5 Class Extends and Implements 36
3.6 Transitive Mutable Field 37
3.6.1 The Transitive Problem of Mutable Field 37
3.6.2 Expanding the Use of RDM on Field 38
3.6.3 Notable Examples 41
3.6.4 Alternative Solutions 42
3.7 Configurable Deep Immutability 49
3.7.1 Problem of Strict Deep Immutability 49
3.7.2 Enjoy Flexibility with Configurable Deep Immutability 51
3.8 Instance Methods for Array 52
3.9 Casting 53
3.10 Polymorphic Qualifier in Inference 53
3.11 Static Inner Classes 54
3.12 Action after Error 54

4 Experiments **56**
4.1 Continuous Integration Tests 56
4.2 Comparison with Glacier 57
4.2.1 Improvements to PICO 58
4.2.2 Comparison with Glacier 59
4.3 Comparison with ReImInfer 63
4.4 Whole Program Inference 63

5 Conclusion and Future Work **65**

References **67**

APPENDICES **71**

A Field Use Table **72**

B Repository of Experimented Projects **76**

List of Figures

2.1 Viewpoint adaptation rules of ReIm 14
2.2 Hierarchy of mutability qualifiers 16

3.1 WF-TYPEUSE's second update, adding the extended-RDM rule 41
3.2 Hierarchy of qualifiers of RDM being supertype 44
3.3 Hierarchy of qualifiers including `@TransitiveMutable` 46
3.4 The formalization for field access with `@TransitiveMutable` 48
3.5 The formalization for static field with `@TransitiveMutable` 48

List of Tables

2.1 Field declarations and abstract state 21

3.1 Truth table of Viewpoint Adapted Subtype 32
3.2 Truth table of Viewpoint Adapted Equality 33

4.1 Entries added to stub file 58
4.2 Tests imported from Glacier 62
4.3 Number and percentage of qualifiers inferred by project 64

A.1 Possible uses of fields under all condition 75

B.1 URLs of the repository exprimented with PICO 76

Chapter 1

Introduction

Ensuring the functionality of programs to be correct is of vital importance in software development, but immutability is frequently cited as a source of bug and security vulnerabilities [12] [10] [13]. Since human errors are inevitable, multiple sophisticated safeguard methods, including static analysis, have been studied [9]. While some modern programming languages have built-in type systems to check some errors, their built-in type systems often only cover basic language requirements, but they are not expressive enough to detect more potential bugs in the code [26].

For example, in Java, the built-in type system can capture many errors by performing static analysis during compilation, such as assigning a string to an integer parameter or invoking a method that does not exist [17]. In these two examples, the type soundness is enforced, so the programmer gets the guarantee from the Java compiler that the variable can never get assigned to a wrong type, or a non-existing method can never get invoked. However, this guarantee is not expressive enough to cover all the needs of the programmer, such as enforcing the mutability type of an object or class.

Various solutions have been proposed to enhance type systems. One of the main solutions is the pluggable type system which solves the problem that the built-in type systems being not expressive enough. Pluggable type systems are lightweight plugins to the conventional type system of the language that focus on one or more aspects of type [26]. For example, a pluggable nullness type system for Java would only concentrate on the nullness type. The type system can either be static, dynamic, or gradual where only partial type information is given before the compilation [8].

By only enforcing a smaller subset of the type properties, the development of a single checker can be much easier than a monolithic type system, which supports all interesting

type properties. Additionally, a single checker can provide the option for developers of only enabling or disabling certain type systems if needed.

The Checker Framework is such a framework for developing pluggable type systems for Java [7] [26], and is used in this thesis. Type system designers can easily implement their type system with the functionalities that the Checker Framework provides, and also use the existing built-in type systems in the framework, such as nullness and interning type system, to build a higher-level type system based on these.

To reduce the qualifier burden of the developers who use a type system, Checker Framework Inference can provide functions to implement an inferrable type system [1]. An inferrable type system means that it can infer a possible type for a type location where the type is unspecified by the user. The inference tool will traverse all the source code of the whole program, and try to find a valid solution to the types. By using inference, the source code can be fully or partially unannotated, effectively reducing the qualifier burden on a large project [15].

Immutability is often partially supported in programming languages. For example, in Java, the programmer can set a field to final, and if all the fields are final, the class can be considered as shallow immutable. If the fields of all fields are also final transitively, the class is considered deep immutable. However, the immutability issue is still frequently cited as a source of bugs [12][13][10], which means that the built-in approaches are not sufficient.

If the immutability rules of objects are not enforced correctly, it would cause various issues in the software development. For example, a hash map data structure relies on the immutability of the key [4]. If the fields of the hash key are modified during the execution, the map will not recognize the key, leading to a bug that the value associated with the key is unreachable. Moreover, when sharing the object in a concurrent environment, if the immutability of the object is not guaranteed, the problem of race condition can arise, and typically the mutual exclusion lock is used to avoid this problem. However, if an object is guaranteed to be immutable, the need of locking is eliminated for the object.

Naive solutions include making the fields final and ensuring the writable reference is not leaked after full initialization [10], or documenting the mutability property in any kind of documentation, such as a JavaDoc. However, both methods have a great limitation: the first one relies heavily on peer inspection, which has an unacceptable chance of introducing human error in the process. Also, the solution requires additional efforts to update the existing code and may become too restrictive. The second method is flexible and easier to implement compared with the first method, but it does not have any enforcing effort, introducing more potential for human error compared with the first method. A careless

programmer is very likely to overlook the instructions in the document and mutate the object. As a result, failures arise in other components which will be hard to trace.

Previous research presented various pluggable type systems for the mutability type, enforcing the immutable rules in the compiler while ensuring user-friendliness. Such type systems include Jarari [29], ReIm [21], and Glacier [12]. While the immutability rules are enforced, the flexibility of the code is more or less affected.

To achieve both enforcement and flexibility, a newer immutable type system, PICO [28], was developed. PICO introduced the receiver-dependent mutability qualifier to make the type checking process receiver sensitive, so the checker will have more context and knowledge of the code. Here is an example for how immutability type could benefit the development:

```
1 @Immutable
2 class ImmutableKey {...}
3
4 class Test {
5     void foo(HashMap<ImmutableKey, Object> map) {
6         map.add(new ImmutableKey(...), ...);
7     }
8 }
```

For the invocation of `map.add`, if the key is not guaranteed to be immutable, it can be a potential source of bug. However, in this code, the key is guaranteed to be immutable by PICO, so the mutation of the key can never be a bug for the code, removing a possibility for debugging to save time.

While the idea of PICO is promising, the formalization neglected certain unsoundness, and the implementation is not compatible with the latest version of Checker Framework.

The focuses of this thesis are on the improvement of PICO to make it more practical, so the immutability type system could be both sound and flexible, eliminating the mutability-related issues while preserving most of the existing structure of the code without major changes. This thesis makes contributions as the following: the thesis

- clarifies the definition of the class bound, which was defined ambiguously;
- presents a refined ordering of the qualifier substitution process involving type variable resolution, qualifier polymorphism, and viewpoint adaption;

- introduces the adapted subtyping operation, which is used heavily in PICO to ensure the flexibility of the receiver-sensitive qualifier;
- revised the mutability definition of class bound, subtyping, casting and enum types;
- presents a solution to a problematic defaulting and qualifier usage rule that would break the immutability guarantee, ensuring the field transitively read-only in mutable classes by default;
- presents a way to ensure deep immutability of immutable classes by default, but still overridable by the user;
- presents a fix to instance methods of the array objects, which cannot be annotated in the previous mechanism; and
- improves the actions during checking and inference to be more friendly to users.

This thesis is organized as follows:

Chap. 2 discusses the background on the pluggable type system, Checker Framework, Checker Framework Inference, and introduces the past works on immutability, including PICO. Chap. 3 introduces the improvements and clarifications made to PICO. Chap. 4 introduces the experiments conducted to the revised PICO and comparisons to the past projects. Finally, Chap. 5 concludes the whole thesis and discusses future work.

Chapter 2

Background and Related Work

2.1 Pluggable Type Systems

For many programming languages, the built-in type system can prevent basic type problems, but often they are not powerful enough to capture enough errors. Some constraints that may be helpful to programmers are often not enforced by the default type systems of languages [26][23].

Using a pluggable type system is a solution to the problem. By using a pluggable type system, the programmer can make decisions for the type by applying qualifiers in a more abstract level where human errors are more unlikely to occur.

For a concrete example, if a programmer assumes a parameter can never be null, they can plug a nullness type system to the compiler, and apply `@NonNull` to the parameter type. During the analysis, the type system will check the invocations of the method, if the argument got null on any invocation, the type system will warn the user. If no warning or errors are issued by the type system, the programmer has a guarantee that the parameter is never null. Compared with manual inspection on all the uses of the method or using JavaDoc, simply applying a qualifier on the target can minimize the chance of human error.

A pluggable type system should both be pluggable and a type system:

- *Pluggable*: the type system should be an *optional* part of the project, which means that the type system can be disabled or completely removed from the execution environment. For example, the type analysis is only performed in the development environment, and after all problems related to the type system is resolved, the type system is not needed in the production environment.

- *Type system*: the system itself should be a complete type system with rules, types, and qualifiers, unlike many lint tools that only perform simple syntactic checks based on string or regex matching. That means that the type system should "understand" the elements of the language.

An optional type system can be static or dynamic. For static type systems, the check occurs before the runtime, during the compilation [16]. A fine-tuned static type system can capture all errors before the program runs and have no runtime overhead, granting a guarantee that the program will always behave within the limitations by the qualifiers, and no runtime errors will happen. But static check cannot capture all problems, such as downcasting and the boundary of the system where objects come from an unknown source. When the knowledge is limited, the static type system can only issue partial results, and making reasonable assumptions based on the type system. For the dynamic type systems, the system will inject checks to the code in runtime, so there will be additional overheads to perform the check on the type usage. However, the concrete runtime value is available to the type system, eliminating the possibility of lacking the knowledge of the type usage.

Static and dynamic type systems are not mutually exclusive. For example, granular type system is a hybrid of static and dynamic, taking the advantage of both kinds of type systems. A granular type system can perform a static type check on the analysable parts of the code, and it can perform dynamic check on the parts of code which require runtime information to check. A type system can also have a dynamic component, and Javari [29] is such an example: it performs a static check on most uses of type and dynamic check on downcasting where the actual type is only determinable during runtime.

One drawback of a pluggable type system is that the developers must explicitly put qualifiers to convey their assumptions to the type system. If the scale of the project is not large, or the qualifiers are applied at the beginning of the development, it should not be a problem. However, applying qualifiers on a large-scale existing project is a real problem: using the type system will introduce unacceptable entry effort. While the defaulting can help with this problem, e.g., using `@NonNull` as the default qualifier can reduce accidental error, but the defaulting does not always reflect the intention of the developers.

Type inference can solve this problem [15], and reduce the annotating effort to an acceptable level. During inference, the code will be automatically annotated with the existing formalization of a type system. Finally, the output will contain a possible solution of qualifiers applied to the code that is consistent with the type rules. When there is a contradiction and the inference failed, it means parts of the code do not comply with the type rules and needs fixing, or the code is too tricky to be covered with the inference tools.

To make the output more meaningful to the programmer when there exist multiple choices on a type use position, the inference tool can have a preference, often the most restrictive qualifier. Take the nullness type system for example, since the built-in behavior of Java language is to allow the type to be null, without preference, the inference result could be `@Nullable` on every type location. If the inference tool prefers the `@NonNull` qualifier when possible, the result will be more useful to the programmer because it reveals that the type use can never be null.

There are several approaches to infer a qualifier on a type usage: by removing the wrong option [21], or by specialized constraints of a type system [22], or by generic constraints with SAT/SMT solver [15].

2.2 Checker Framework and Inference

2.2.1 Checker Framework

The Checker Framework [26] is a framework for implementing pluggable type systems for Java. With the aid of the Checker Framework, implementing a new pluggable type system will be an easy job, with the low-level details abstracted away, for example, traversing the abstract syntax trees, applying defaults, and generating common constraints.

The Checker Framework also contains a collection of checkers that can be used by another checkers as a sub-checker. For example, any type of system can call the initialization checker inside the Checker Framework before the check runs, and will be able to get the initialization status of the types which can be combined with the type rules.

The Checker Framework is implemented as an annotation processor of the Java compiler, so the checking process happens during the Java compilation after the source code is parsed and the Java compiler trees are provided to the annotation processor. Checker Framework will handle most of the Java compiler API for the checker developer to facilitate the checker developing process.

During the process of compiling, Checker Framework is firstly invoked by the Java compiler, then Checker Framework will invoke the checker used by loading the class of the checker. The checker traverse through the syntax tree and try to find any errors based on the qualifier and the type rules after applying defaults. If any errors found, the checker will report them and the build fails. Otherwise, the build succeed.

The Checker Framework has built-in support for several functionalities. For example,

the data-flow framework for flow-sensitive refinement [2], and qualifier polymorphism. Here is an anatomy of a Checker Framework type system:

- `Checker` class: the "entry point" of the checker, because the framework will load this class of a checker after some pre-procedures. Contains meta configurations of the checker, such as all of the components, supported qualifiers, and supported options, etc. With the automated loading of the classes of components in Checker Framework, all the qualifiers in the `qual` package will be automatically loaded as the supported qualifiers, and the framework can automatically import the components following the naming convention of Checker Framework. Thus, a checker does not need to override most of the configurations in many cases.

- Qualifiers: Checker Framework uses Java 8 annotations as qualifier, so the qualifiers have to be declared as annotations. Also, the designer can specify the hierarchy of the qualifiers here by using meta qualifiers provided by Checker Framework, and the framework will build the hierarchy for later use. The defaulting rules are also implemented here by using the meta qualifiers indicating which type should the qualifier be the default.

- `AnnotatedTypeFactory` class: defines how a language construct maps to a type, the defaulting rules, and types for literals. For example, converting a `MethodInvocationTree` to a `AnnotatedExecutableType`. The designer can alter the way a checker determines the type of a syntax tree.

- `Visitor` class: the visitor traverses the syntax trees. The designer will implement the type rules here, and just override the corresponding methods for the language construct. For example, the designer will implement the rules for method invocation by overriding the method `visitMethodInvocation`. The framework has already implemented generic type rules, such as the subtyping rules in assignments, so the designer does not need to take care of them.

- `Validator` class: the validator of the types. Different from the `Visitor` class that traverse the syntax trees, this class is called to validate the types. So the type rules that is applied globally on all uses of the types is implemented here. For example, if a qualifier is only intended for internal use on a certain type, a rule can be implemented here, checking the explicit qualifiers on the type.

- `Transfer`, `Value` and `Analysis` class: since Checker Framework supports dataflow framework to perform flow-sensitive refinement, such as refining the type of a local

variable to a more concrete type from the assignment context. If the default rules are undesirable, the designer can override them in this class.

- `ViewpointAdapter` class: contains the rules for viewpoint adaptation. Viewpoint adaptation is a function that maps from the type itself and the receiver type to another type. It allows the qualifier to express the relative type to the receiver, for example, for a ownership type system, a qualifier on the type can be used to express that the reference has the same ownership status with the receiver, or is owned by the receiver [20].

The type system designer can also overrides other components if they are unsatisfied with the built-in rules of Checker Framework.

2.2.2 Checker Framework Inference

Checker Framework Inference is a framework for the type inference of unannotated or partially-annotated code. As a inference tool, it can infer a possible assignment of qualifiers for the code that complies with the defined type rules, or reach a inference failed state if no possible solution can be inferred, often caused by the code contains conflict to the type rules.

Checker Framework Inference is a generic inference tool for all type systems. In order to achieve that, it supports several high-level constraints for the designer to define the type rules:

- Subtype ($q_1 <: q_2$): qualifier q_1 should be a subtype of q_2.
- Equality ($q_1 = q_2$): qualifier q_1 and q_2 should be the same.
- Inequality ($q_1 \neq q_2$): qualifier q_1 and q_2 should be different.
- Comparable ($q_1 <:> q_2$): qualifier q_1 and q_2 should have a subtyping relation, i.e., $q_1 <: q_2$ or $q_2 <: q_1$.
- Combine ($q_3 = q_1 \triangleright q_2$): qualifier q_3 should equals to the viewpoint adaptation result of q_1 and q_2.
- Preference ($q \sim= c$): qualifier q should equals to c whenever possible. c is a constant.

- Implication ($c_1c_2...c_{n-1} \Rightarrow c_n$): if constraint $c_1 \land c_2 \land ... \land c_{n-1}$ hold, c_n should also hold. This is a composite constraint.

If the provided operations are not enough, the designer can always introduce new operations to the system.

Checker Framework Inference uses the SAT or SMT solver to solve the constraints, so all the type constraints the designer added to the system will result into CNF or SMT formulas, translated by encoders. The constraint can be both enforced, or breakable but preferred [30].

In order to associate the variables in the formulas with the type use location, the `Annotator` classes are used to put a special qualifier into the code: `@VarAnnot`, which contains the ID of the variable in the solver. So, after the solver resolves a model, the values in the solution can be inserted back. For example:

```
1 @VarAnnot(4)
2 class MyClass extends @VarAnnot(5) Object {
3     @VarAnnot(6) Object foo(@VarAnnot(7) MyClass this, @VarAnnot(8)
   ↪  Object obj) {
4         return @VarAnnot(9) null;
5     }
6 }
```

The annotators will put the variable qualifiers in the code, so when applying the type rules in the visitors, in fact a constraint is generated between the operand variable qualifiers instead of the real ones. Note that the IDs in the example will be different among checkers, so the values are just for demonstration.

There are also special variables that do not correspond to a type location, so they are not insert-able to the code after the solver get a model. For example, the constant variables for each qualifier in the type system, or the variables for the viewpoint adaptation result.

After the solver gets a model, Checker Framework Inference will inject the value of every insert-able variables back to the code, so the developer could examine whether the inferred result is desirable.

To support different solvers, Checker Framework Inference provide a way to implement new encoder and runner to the solver. If a developer needs to use a new solver, they only need to create a new encoder to call the solver's API without changing the rest of the framework.

2.3 Mutability Type Systems

The goal of a successful mutability type system is to reduce the programmer's effort in making a type immutable. Ideally, the syntax of the type system should be simple enough to minimize the possibility of human error, and the system should automatically enforce the mutability rules on the uses of the type without doing major changes on the type, such as making all fields final. Also it should detect possible errors that is not directly supported by the language, unlike the final in Java, such as leaking the reference of the type in constructor when the reference is still mutable.

Designing a mutability type system requires multiple design choices among several dimensions. Earlier research on the syntax of immutability identified multiple different dimensions of immutability [13][10], and the key dimensions include:

- Assignability vs writability: assignability refers to the assignment to variables, and the writability refers to the re-assignment of the field of the variables. Whether the immutability should prevent both in all circumstances is an important design decision.

- Object or class granularity: if the checker supports a granularity of object, the mutability restriction could be applied differently on each object of the same class, while a granularity of class will enforce the same mutability restriction on all objects of the same class.

- Transitivity: without transitivity, the mutability restriction applies only on the direct field of the variable, known as shallow immutability, while with transitivity, the mutability restriction applies on all fields reachable from the variable, also known as deep immutability, which is a stronger restriction and provide a safer guarantee. This involves the concept of *abstract state*. Javari's definition of abstract state is: a (part of) the transitively reachable state, that is, the state of the object and all state reachable from it by following references [29]. By default it is the whole reachable state, but removing parts is possible.

- Qualifier polymorphism: similar to the object polymorphism where the invoked method will be dispatched based on the arguments, if the type system supports qualifier polymorphism, it means that one function can accept arguments of different mutability types. While in non-polymorphic type systems the mutability type could have only one choice.

- Static or dynamic enforcement: static enforcement only occurs during compilation, and finds all possible errors without actually executing the program. Thus, this kind of type system will not introduce overhead at run-time, only introducing overhead to compilation, but will not respond to errors caused by certain language features, such as downcasting and covariant array subtypes. While dynamic enforcement will perform the check during run-time, so it will introduce run-time overhead, but sensitive to the dynamic language features that static systems cannot support. An intermediate option could be gradual enforcement, where the enforcement happens not only in compilation, but also in the runtime for the unknown types [8].

2.3.1 Javari

Javari [29] is one of the pioneer projects on the immutability of Java. Javari is a type system based on immutability constraints and supports both static and dynamic enforcement.

Since Javari is constraint-based and supports both static and dynamic enforcement, the dynamic part is not enabled for all constraints. To reduce runtime overhead, dynamic enforcement is only enabled for cases that cannot be handled by static checks, such as downcasting.

Javari provides a transitive mutability guarantee, so it enforces deep immutability. However, the user can still exclude a field from the abstract state explicitly if the mutability type of the field is not cared for by the developers.

Javari works in both class and reference level. The qualifiers for mutability types are:

1. `readonly`: the reference cannot be used to mutate the object it refers to. When applied to class, all its objects are of this type.

2. `mutable`: the reference can always be used to mutate the object.

3. `this-mutable`: the reference is polymorphic, and the mutability type inherits from the receiver. This qualifier is not class-level, and only applicable to references, because there is no receiver in a outer or static inner class definition. For example, if the receiver reference is `readonly`, the mutability type of the current reference (e.g. a field) will also be `readonly`.

Javari explicitly separated the mutability and assignability of a reference. So, a set of assignability qualifier is also available in Javari:

1. `assignable`: the reference is always re-assignable even if its receiver is `readonly`. By using this qualifier, the reference is explicitly removed from the receiver's abstract state. For example, if a field is `assignable`, even if the receiver reference is `readonly`, the field itself is re-assignable.

2. `final` the reference is not re-assignable after the first assignment. Supported by Java.

3. `this-assignable`: the assignability of the reference is polymorphic, and the assignability is decided by the actual receiver.

Note that even if a field is `assignable`, it could also be `readonly`, which means that the field can be re-assigned, but the internal state of the referred object cannot be mutated.

Since customized Java annotation was not available at that time, Javari extends new type modifiers into Featherweight Generic Java for formalization.

2.3.2 ReIm and ReImInfer

ReIm [21] is a static immutability type system with similiar qualifiers to Javari: `mutable`, `polyread`, and `readonly`.

1. `mutable`: the reference can always be mutated, just like a regular Java reference.

2. `readonly`: the referred object cannot be mutated via this reference, and since ReIm enforces transitive immutable, its fields within the abstract state cannot be mutated.

3. `polyread`: the polymorphic qualifier of ReIm hierarchy that enables context sensitivity. Its final value is decided by the actual use of the type, based on the arguments, and resolved by *viewpoint adaptation*.

The hierarchy of ReIm qualifiers is: `mutable` $<:$ `polyread` $<:$ `readonly`.

ReIm also supports receiver sensitivity by performing viewpoint adaptation on field accesses and method invocations. Its viewpoint adaptation rules are shown in Figure 2.1.

From the viewpoint adaptation rules we can learn that the transitive immutable also applies on `mutable` fields. Even if the mutability type of the field is `mutable`, if the receiver is `readonly`, the field access will still become `readonly`. So the user has no way to exclude the field from the abstract state.

$$_ \triangleright \texttt{mutable} = \texttt{mutable}$$

$$_ \triangleright \texttt{readonly} = \texttt{readonly}$$

$$q \triangleright \texttt{polyread} = q$$

Figure 2.1: Viewpoint adaptation rules of ReIm

ReIm does not support the assignability qualifiers, so the assignability of the fields is fixed by the mutability qualifiers.

The polymorphic qualifier of ReIm also has a problem: the notation is different from field access and method invocation, which violated the Uniform Access Principle [24]. While the viewpoint adaptation rules remain the same, the target of the viewpoint adaptation is different between field access and method invocation: for field access, the target is the receiver, for method invocation, the target is the calling context. This makes it harder to understand the type system.

ReIm also offered an inference system, ReImInfer, to automatically infer the possible qualifier of a partially or fully unannotated code. The ReImInfer will start from a default set and keep refining the set by removing the qualifier that violates the type rules of ReIm, so no SAT or SMT solver is involved. If multiple solution exists on a location, ReImInfer gives a preference on `readonly`, then `polyread`, finally `mutable`. The time complexity is claimed to be $O(n)$ but $O(n^2)$ in the worst case.

One interesting point of ReImInfer is that it also provided a tool to inference the purity of a method. By combining method purity and immutability check, the type system can be more complete and easier to use for the user.

2.3.3 Glacier

Glacier [12] is a class-level static immutability type system, designed by an evidence-based approach, implemented with the Checker Framework.

One main consideration of Glacier is the easiness to use. For simplifying the syntax, this type system only supports class-level immutability for most classes, which means there could be no qualifiers on the use of a class. This design will not only help the user learning faster but also simplifies error messages. However, the write-protection qualifier `@ReadOnly` is still applicable on the reference to write-protect an object of the mutable class. The only

exception of class-level immutability is for objects in which the class declarations are not available in any Java code, including the Java runtime classes, e.g. arrays in Java[1], object-level qualifiers are still available. Otherwise, there is no way to express what the mutability type could an array object be.

The Glacier supports these qualifiers:

1. `@MaybeMutable`: the direct objects of the classes are mutable, and the subclasses of the class can be `@Immutable`. This design will solve the *fragile base class problem* [25]. Since the superclass can make no assumption on the mutability of itself because of a possible immutable subclass, the qualifier means that the object of the class *maybe* mutable, so no guarantee of mutability can be made.

2. `@Immutable`: the objects of the class and its subclasses are guaranteed to be immutable. The subclass of a `@Immutable` class can only be `@Immutable`, ensuring the reference must be immutable.

3. `@ReadOnly`: same to other systems. Reference bearing this qualifier will not be used to mutate the object it referred. Only usable on reference-level.

4. `@GlacierBottom`: only used on array objects to declare that the array is assignable to any reference.

To reach the maximum simplicity, Glacier does not support the assignability dimension. That means all the fields of an immutable class can only be final. Even the `final` modifier is not on the field, the field is not re-assignable after constructor invocation. And since the mutability qualifiers are class-level, the transitivity is enforced, and a user has no way to exclude a field from the abstract state, unlike Javari. This ensures deep immutability and makes the code safer, but sacrificed the flexibility.

Glacier has no runtime components, so it cannot check downcasting, but still provides warnings if the operation is possibly dangerous.

Glacier does not support qualifier polymorphism, so every qualifier on the method signature has its own type definition, the user has to declare multiple methods if the operation will be performed on both `@Immutable` and `@MaybeMutable` objects.

To ensure the easiness to use, Glacier conducted extensive user studies to ensure the user can use the type system smoothly, and the result shows that compared with vanilla Java, Glacier makes it very easy to make a class immutable.

[1]Java arrays are implemented in JVM. A pseudo class type is provided for the compiler without any classes file associated.

2.4 PICO: Practical Immutability for Class and Objects

PICO allows developers to easily define the mutability type for a class with type qualifiers [28]. Each qualifier on the class declaration represents a mutability type for the class: `@Mutable`, `@Immutable`, and `@ReceiverDependentMutable` (abbr. `@RDM`). Additionally, PICO also supports `@Readonly` on a reference, and `@Bottom` on special uses of null types and the lower bound of the type variable. For the lattice of PICO qualifiers, since every object can be read-only, `@Readonly` is the top qualifier that can apply to any reference to a type. And as the name indicates, `@Bottom` is for the bottom qualifier, based on the assignability of null. A special qualifier `@PolyMutable` is also used in PICO to support qualifier polymorphism. Figure 2.2 gives a clear picture of the hierarchy of PICO qualifiers.

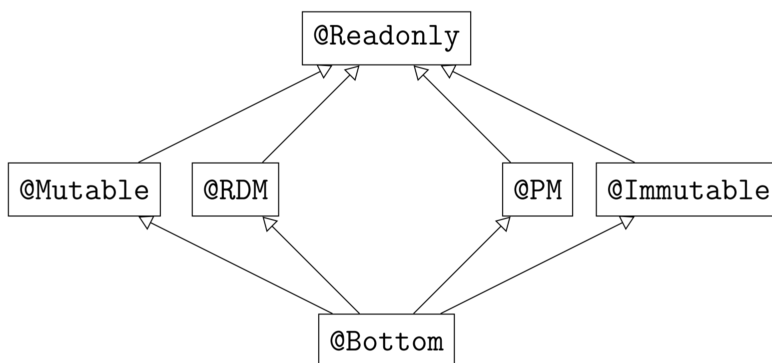

Figure 2.2: Hierarchy of mutability qualifiers

Similar to the other mutability type systems, PICO uses the `@Immutable` qualifier to indicate that all the objects of the class are immutable as a class qualifier, or this object is immutable as an object qualifier. The meaning of `@Mutable` qualifier is also straightforward: the objects of the class must be mutable. The user of the object should be aware that even the object is not mutated within the scope, it could be modified via alias by other code outside of the scope. Since their uses are not compatible with each other, they are not subtypes of each other.

One major difference of PICO from other mutability type systems is the context-sensitivity. PICO supports receiver-context sensitivity which is implemented by `@RDM`, and assignment-context sensitivity which is implemented by `@PolyMutable`.

Receiver-context sensitivity For `@RDM`, the qualifier means the type is receiver-context sensitive, i.e. the type will be affected by the receiver, so the class is neither `@Mutable` or `@Immutable`, but capable of having objects of both kind, depending on the receiver. The meaning of `@RDM` differs by qualifier locations:

- As a class instantiation bound: means the qualifier on the use of the type can be any type as a result of being receiver-context sensitive, so it is similar to the `@MaybeMutable` in Glacier for this usage. This is different from `@Immutable` and `@Mutable` which limits the uses to be only their own types or `@Readonly`. Since the usage may be an `@Immutable` type, PICO will enforce some of the rules for immutable type on RDM class declarations, e.g., all fields should be explicitly initialized.
- On a field declaration and method declaration: the qualifier means that when in the actual use, i.e. the field access or member method invocation of a concrete object, this qualifier will be resolved to the receiver's type.
- On a constructor declaration: the qualifier means that the invocation of the constructor can be one of the bound qualifier: `@Immutable`, `@Mutable`, and `@RDM`. By doing so in the invocation, the mutability of the `@RDM` object is decided by the qualifier provided. If `@Immutable` or `@Mutable` is provided, it means that the receiver-sensitive feature is explicitly disabled on the object created, so its type no more depends on the receiver.

Receiver-context sensitivity helps with improving the flexibility of the type system. Since `@Mutable` and `@Immutable` in PICO is strictly mutable or immutable, `@RDM` means that the class could be either type, which is useful when a common class is needed, and both kinds of instantiation are required. For example, a utility class `ByteArray`. In our code examples, qualifiers in inline comments are the default qualifiers without the need to be explicitly written.

```
1  @ReceiverDependentMutable
2  class ByteArray {
3      private char[] data;
4      public void mutate(@Mutable ByteArray this) {...}
5      // ...
6  }
7  @Mutable
8  class MutableUser {
```

```
9       /*@ReceiverDependentMutable*/ ByteArray bytes;
10      // ...
11  }
12  @Immutable
13  class ImmutableUser {
14      /*@ReceiverDependentMutable*/ ByteArray bytes;
15      // ...
16  }
17
18  class Main {
19      void foo(/*@Mutable*/ MutableUser mu, /*@Immutable*/ ImmutableUser
         ↪  iu) {
20          mu.bytes.mutate();
21          iu.bytes.mutate();
22      }
23  }
```

Although the same type `@RDM ByteArray bytes` presents in the two class `MutableUser` and `ImmutableUser`, the meaning of the type is completely different because of receiver-context sensitivity. For the use of the field of `MutableUser` in `mu.bytes`, after taking account of the recevier, the type is `@Mutable ByteArray`, while in the use of `iu.bytes`, because the receiver is `@Immutable`, the type becomes `@Immutable ByteArray`.

Assignment-context sensitivity Similar to the receiver-context sensitivity, sometimes the type also needs to depend on the assignment context, i.e. both of the types on the left and right-hand side of the assignment are required to determine the final type.

This code example demonstrates a use of assignment-context sensitivity.

```
1   @ReceiverDependentMutable
2   class ByteArray {
3       // creates instances of any type needed by the assignment context
4       static @PolyMutable ByteArray createAny() {...}
5   }
6
7   @Mutable ByteArray mutableBytes = ByteArray.createAny();
```

In the code above, the left-hand side of the assignment is also used to decide the final type of the method invocation `ByteArray.createAny()`. The `@PolyMutable` qualifier on the method declaration is a polymorphic qualifier, and it will be resolved to the most suitable type based on the actual invocation. When used on the returning type, it means that the assignment context is also used to define the final type. In this case, the `@PolyMutable` is only resolved based on the assignment context, which requires the type to be `@Mutable`. In PICO, without additional code, the type of `ByteArray.createAny()` will be resolved to `@Mutable ByteArray`.

Not only the real assignments, pseudo assignments is also taken into account. For example, if the `createAny()` method is invoked on a parameter of another method which requires a `@Immutable ByteArray`, when resolving the method invocation, the pseudo assignment of `@Immutable ByteArray` is also used, causing the invocation to `createAny()` results a `@Immutable ByteArray` return type.

Class and Object Level Mutability PICO supports both class and object-level mutability. That means that the mutability of an object can be different from the class declaration. One example of this is the `@RDM` classes, which can have objects created as `@Mutable` or `@Immutable` by invoking the corresponding constructor.

For `@Mutable` and `@Immutable` classes, although the constructor type can only be the same with the class-bound, the references to the objects can still be `@Readonly`, which means the object level mutability still applies.

Separation of Assignability and Mutability Assignability is different from mutability. For example, consider a field that is mutable by non-pure methods, but not re-assignable. Or a field that is immutable, preventing any change, but allowing the field to be re-assigned to another immutable object. To support such type rules, PICO also supports a assignability qualifier `@Assignable`. Together with the Java's `final`, the assignability dimensions are:

- `@Assignable`: the field is always re-assignable.
- `final`: the field is not re-assignable.

If no assignability qualifier presents, the assignability of the field is defined by its receiver. If the receiver type is `@Mutable`, the field is considered as `@Assignable`, and if the receiver type is `@Immutable`, the field is considered as `final`.

Circular Initialization of Immutable Objects While stricter checkers will not allow the reference to `this` leaked outside of the constructor, such as Error Prone [3], PICO allows an immutable object to be constructed circularly. PICO relies on an initialization sub-checker provided by the Checker Framework [27] to determine the initialization status of an object.

This code example is inspired by *Freedom before Commitment: A Lightweight Type System for Object Initialisation* [27], demonstrates a circular initialization process.

```
@Immutable
class SingleNodeList {
    @Immutable ListNode head;
    @Immutable SingleNodeList() {
        head = new ListNode(this);
    }
}
@Immutable
class ListNode {
    @Immutable SingleNodeList list;
    @Immutable ListNode(@UnderInitialization SingleNodeList list) {
        this.list = list;
    }
}
```

During the initialization process in `List()`, the receiver type `this` has the qualifier of `@UnderInitialization` added by the initialization type system, so passing `this` to `ListNode` is valid. After the `List` constructor finishes, both instance of `List` and `ListNode` turn into fully `@Immutable` objects, preventing any further mutations.

Method Purity Impure methods can mutate the receiver object or arguments by side-effects [19]. By ensuring the immutability of `@Immutable` objects and `@Readonly` references, PICO allows adding a qualifier to the receiver and parameters. For example, if the receiver is `@Immutable`, the method cannot be invoked on a `@Mutable` object. This is helpful with `@RDM` classes, for example, making the setter methods only invocable by its `@Mutable` instances. By applying `@Readonly` to the receivers and parameters, it means that the method is side-effect-free, and can be safely invoked by any objects with any arguments.

Exclude Field from Abstract State For immutable classes and objects in PICO, the fields are by default protected by the immutability guarantee: the fields cannot be re-assigned or mutated by a non-pure method that has a side effect. However, if the user needed, they can override the behavior by using `@Assignable`, `@Mutable` or `@Readonly` on the field, and exclude these fields from the abstract state, as shown in Table 2.1. "X" means that the field is not a part of the abstract state, and "O" means that the field is a part of the abstract state.

	`@Mutable`	`@Readonly`	`@Immutable`	`@RDM`
`@Assignable`	X	X	X	X
`final`	X	X	O	O
`@RDA`	X	X	O	O

Table 2.1: Field declarations and abstract state

By using `@Assignable`, the field is always allowed to be re-assigned. By using `@Mutable` and `@Readonly`, the field may have an alias outside the immutable class which can be used to mutate the field, and this is out of the control of the immutable class. Note that for `@Readonly`, although the field is not mutable via the field within the class, the underlying mutability type is unknown, so there is a possibility that the field being a `@Mutable` type and be mutated outside the class.

Chapter 3

Improvements to the PICO Type System

Although PICO is a mutability type system more flexible than many of the existing solutions, when we updating the checker, certain design issues of the checker are revealed. In this chapter, we will introduce the improvements we made for PICO to make it more flexible, or to fix the bad designs.

3.1 Class Bound Use

3.1.1 Definitions of Different Bounds

There are two kinds of bound in PICO:

- Upper bound: refers to the top qualifier of the hierarchy, the `@Readonly`. In a lattice, this is called the top.
- Class instantiation bound: refers the qualifier on the class declaration, and only have three options: `@Mutable`, `@Immutable` and `@Readonly`, which will be referred as "bound qualifier" later in this thesis.

The meaning of a class instantiation bound is that every concrete object created should be with the same qualifier as the class instantiation bound, with the exception that class

with `@RDM` instantiation bound can have all three kinds of objects of `@Mutable`, `@Immutable` or `@RDM`.

Note that a reference is not equivalent to an object. So one non-bound qualifier is allowed on the reference, the `@Readonly`. Since `@Bottom` have special meanings in PICO, it is not allowed on most type usages. The validity rules of PICO rely heavily on the instantiation bound.

Later in this thesis, we will use "`@q` class" to refer to a class with the instantiation bound of `@q`.

3.1.2 Class Bound Handling

For Checker Framework Inference, during the generation of slots on class bounds, the class for annotating the code, `VariableAnnotator` will generate a `ExistentialVariableSlot` on the slot of declaration annotation. This existential slot will act differently whether a qualifier is present on the position or not: if presents, adding a set of constraints to the solver, if not, adding another set of constraints to the solver.

This design is to ensure the slot generation works with all kinds of checkers, and the checker which acts differently when an explicit annotation on the class declaration will benefit most from this approach of slot generation.

But for PICO, *every* class have a declaration annotation, known as the instantiation bound. If an explicit annotation is missing in the code, during modular type check, a default one, `@RDM`, is applied. So, in PICO, if a class does not have a declared annotation after defaulting, it is considered as a kind of internal error.

Moreover, the solver PICO uses, `MaxSAT`, is newly ported to checker framework, and does not support all kinds of constraints. Since the existential slot is not used in other parts of PICO, we did not implement the encoder for this kind of slot to reduce the implementation burden.

Finally, if we already know that one existential slot in PICO must exist, replacing that with a regular variable slot will reduce the overhead of low-level constraint encoding.

Based on the discussion above, we overridden `VariableAnnotator` in Checker Framework Inference to replace the `ExistentialVariableSlot` with `VariableSlot` during the slot generation on a class declaration annotation.

3.1.3 Anonymous Class Bound in Inference

For an anonymous class without the keyword and the name, the user cannot assign one annotation like a regular class which annotation is located before the name. But since the new operator is the only use of the anonymous class, there is no need to make a distinction between the instantiation bound and the only use:

- For `@Mutable` and `@Immutable` class, the only valid invocation of constructor is their instantiation bound, so they are the same.
- For the trickier `@RDM` class, although the constructor can have different types and the invocation can be all kind of the bound qualifiers, an anonymous class cannot have a named constructor, preventing the creation of a constructor of different type. And PICO permits a class other than `@RDM` extending a `@RDM` class, given the type is specified on the extends clause. For the clause, it cannot have any option other than the qualifier on the new operator.

This code examples demonstrates the equivalent regular class for an anonymous class:

```
1  @Immutable RDMInterface rdm = new /*@Immutable*/ RDMInterface() {...};
2
3  @Immutable
4  class anonymous implements /*@Immutable*/ RDMInterface {
5      /*@Immutable*/ anonymous() {...}  // generated implicit constructor
6  }
```

As we can see from the code example above, all the locations: class declaration bound, implements clause and return type of the constructor, is the only option that makes sense[1]. So, it cannot make a distinction between the instantiation bound and the only use.

However, `VariableAnnotator` will generate two slots on the two locations: the new clause, and the anonymous class, and both slots are insertable, meaning that they result in an annotation on the location. The problem is that the anonymous class does not accept an annotation, and if the slot on the anonymous class is inserted into the inferred code, resulting in something like this:

[1]The PICO syntax also permits when the class declaration bound and implements clause are both `@RDM`. But an `@RDM` class does not make sense when all the constructors only return `@Immutable` type.

```java
@Immutable RDMInterface rdm = new /*@Immutable*/ RDMInterface()
    @Immutable {...};
```

Undoubtedly, this is a bug in the `VariableAnnotator`. To fix this, two potential solutions are discussed:

- Keep the two slots, and make the anonymous class slot not insertable (hidden from code). However, the slot still get inferred. For the checkers that the types on the new clause and the anonymous class bound are guaranteed to be the same, add an equality constraint between the two slots.

- Only generating one slot for the anonymous class, and use that for both the type of new clause and the anonymous class bound. Insert the slot to the new clause only.

After extensive discussion, we reached the conclusion that no other checker will ever requires the qualifiers on new clause and the anonymous class bound to be different, since a qualifier cannot be put on a anonymous class after all.

For the implementation, since the enclosing tree is always visited before the inner tree (otherwise, there is no way for the visitor to know there is a inner tree), a slot should be generated on the enclosing new class tree before visiting the anonymous class declaration tree. Based on this assumption, when generating the slot for the anonymous class, the `VariableAnnotator` gets the slot on the enclosing tree. For the anonymous class, the enclosing tree is guaranteed to be a new class tree. After getting the slot, the annotator will register this slot as the class bound of the anonymous class, instead of generating a new slot.

3.2 Polymorphic Substitution and Type Variable Resolution

3.2.1 Introduction to Polymorphic Qualifiers

Java supports polymorphic methods by overloading, but the polymorphism does not apply to the qualifiers. That means that the object of the same class but different qualifiers are regarded as the same type. That means that such code cannot compile:

```
class PolyClass {
    void foo(@Mutable Object obj) {...}
    void foo(@Immutable Object obj) {...}
}
```

The signatures of the two methods are the same, and the mutability qualifier is ignored, so the method signature is duplicated and causing an compilation error.

To address this problem, Checker Framework supports qualifier polymorphism by adhoc polymorphism: the checker designer can assign a qualifier as the "polymorphic qualifier", and use the qualifier on the method signature. When the method invoked, the qualifier will be resolved to the most fit variant of the method, as if there are different versions of the method. For example, when using the polymorphic qualifier, `@PolyMutable` in PICO, the `PolyClass` above can be changed into:

```
class PolyClass {
    void foo(@PolyMutable Object obj) {...}
}
```

When the method is invoked with a `@Mutable Object`, the signature will be resolved to `void foo(@Mutable Object)`, satisfying the requirement.

Even if unlike the "real" method polymorphism, the qualifier polymorphism cannot have different method bodies for each version of the method, it still very useful in many conditions to make the checker more flexible.

3.2.2 Polymorphic Substitution and Type Argument Resolution

Qualifier polymorphism in Checker Framework is implemented by qualifier substitution: before checking the method invocation with the declaration signature, the polymorphic qualifier is compared with the invocation qualifier, computing the most fit qualifier [2] based on the invocation, and finally the polymorphic qualifier is replaced with the computed most-fit qualifier.

[2] The least upper bound (LUB) of all polymorphic qualifier positions. The details is out of the scope of the thesis, but can refer to the manual if interested https://checkerframework.org/manual/#qualifier-polymorphism

But polymorphic qualifier substitution is not the only substitution process of computing a method declaration type, the other one is type argument resolution. Whenever Checker Framework checks a function invocation with type arguments, it will replace the type parameter in the method declaration with the actual type arguments before checking.

When combined with generics, in which step should the polymorphic qualifier be substituted becomes a problem. The Checker Framework once perform the polymorphic qualifier substitution before resolution of type arguments, but this ordering caused a concerning issue [3]:

```
class PolyNullTest {
  void foo(List<@PolyNull Object> l) {
      l.add(null);  // error not reported
  }
  void test(List<@Nullable Object> n, List<@NonNull Object> nn) {
      foo(n);
      foo(nn);  // may lead to NPE: see the discussion
  }
}
```

`l.add` impose a danger that adding a null reference to a list that only accepts `@NonNull Object`, for example, passing `nn` as the parameter of method `foo`. Since `nn` is a list of `@NonNull` object, the `null` inserted by `foo` may cause a `NullPointerException`.

The reason for this false negative is that when performing the type argument substitution, the polymorphism qualifier on the type argument is introduced to the declaration type of the method which is later substituted to `@Nullable`.

1. The declaration signature: `boolean add(E e)`
2. After type argument resolution (`E` → `@PolyNull Object`):
   `boolean add(@PolyNull Object e)`
3. After polymorphic qualifier substitution (`@PolyNull` → `@Nullable null`):
   `boolean add(@Nullable Object e)`

[3] https://github.com/typetools/checker-framework/issues/2432

The `@PolyNull` is not solvable because the origin of the qualifier is not from the context of the method, but from the parameter of the method. Since the parameter of the method is indeterminable until the actual invocation, inside the method the checker still should make a safe assumption about the qualifier, and keep it in its place.

Based on the discussion above, the key to the fix is to only substitute the polymorphism qualifier from the context of the method. So, the fix to the substitution problem should be switching the order of the process:

1. First perform the polymorphic substitution, avoiding the substitution of qualifier from parameters when the type argument remains unresolved.
2. Then perform the type argument resolving.

For the `PolyNullTest` example, the process of substituting becomes:

1. The declaration signature: `boolean add(E e)`
2. After polymorphic qualifier substitution (no qualifier): `boolean add(E e)`
3. After type argument resolution (`E` $\rightarrow$ `@PolyNull Object`): `boolean add(@PolyNull Object e)`

As shown in the result, the polymorphic qualifier that is from the parameter get preserved. And from our experiments, the fix is compatible with the rest of the rules in Checker Framework.

3.2.3 Polymorphic Substitution and Viewpoint adaptation

Very similar to type parameter resolution, viewpoint adaptation may also introduce a polymorphic qualifier which is not substitutable. For example in PICO:

```
1  class Clazz {
2      @PolyMutable Object echo(@ReceiverDependantMutable Clazz this,
       ↪  @PolyMutable Object o) {
3          return o;
4      }
5
```

```
6	    void test(@Immutable Object o, @PolyMutable Clazz clazz) {
7	        @Immutable Object e = clazz.echo(o);  // should be fine
8	    }
9	}
```

Previously, the viewpoint adaptation is performed before the polymorphic substitution, so the method invocation went though these steps:

1. The declaration signature:
   `@PolyMutable Object echo(@RDM this, @PolyMutable Object o)`
2. Viewpoint adaptation (`@RDM` → `@PolyMutable`):
   `@PolyMutable Object echo(@PolyMutable this, @PolyMutable Object o)`
3. Polymorphic substitution (`@PolyMutable` → LUB(`@PolyMutable`, `@Immutable`) = `@Readonly`): `@Readonly Object echo(@Readonly this, @Readonly Object o)`
4. No type parameters declared, type argument resolution skipped.

So the `@PolyMutable` introduced by viewpoint adaptation should not be a part of the polymorphic substitution. In order to solve the issue, we adopted the same approach of re-ordering: switch the ordering of viewpoint adaptation and polymorphic substitution. After the change, the processing steps become:

1. The declaration signature:
   `@PolyMutable Object echo(@RDM this, @PolyMutable Object o)`
2. Polymorphic substitution (`@PolyMutable` → `@Immutable`):
   `@Immutable Object echo( @PolyMutable this, @Immutable Object o)`
3. Viewpoint adaptation (`@RDM` → `@PolyMutable`):
   `@Immutable Object echo(@PolyMutable this, @Immutable Object o)`
4. No type parameters declared, type argument resolution have no effect.

After the process, the comparison between the declaration signature and the invocation type should be fine, thus the check is passed.

It is worth noting that an alternative method previously applied in PICO is that first replacing all `@PolyMutable` with a new qualifier, `@SubstitutableMutable`, which is a direct subtype of `@Readonly`. So during the viewpoint adaptation the newly introduced polymorphic qualifiers remains `@PolyMutable`. Then during the polymorphic substitution, only `@SubstitutableMutable` qualifiers are regarded as the polymorphic qualifier and get substituted.

This approach requires a new qualifier, making the system more complex. Since we now have a better solution, we are removing the `@SubstitutableMutable` qualifier and the replacement process in PICO.

3.3 Viewpoint Adapted Subtype

3.3.1 The Issue of Formalization

`@RDM` is a very special qualifier in PICO. In many cases, when a checker validates of a use of type itself, it should compare the use qualifier with the qualifier upper bound of the type. If the use is below the upper bound, then the qualifier on the type is valid, and the rest of the type is typically handled by Java compiler. For PICO, the upper bound is `@Readonly`, but one difference is that it also requires a lower bound of the use qualifier, the instantiation bound of the class: `@Mutable`, `@Immutable`, or `@RDM`.

But for `@RDM`, since the use could become any type after viewpoint adaptation and constructor invocation, the upper bound check is not enough. So for `@RDM`, we cannot apply the subtype constraint on the use of the type, otherwise PICO cannot typecheck a `@Mutable` or `@Immutable` use of a `@RDM` class.

Previously, when dealing with the validity of the use of a type, PICO uses such a constraint to satisfy `@RDM`:

$$q_{bound} = \texttt{@Mutable} \Rightarrow (q_{use} \neq \texttt{@Immutable} \land q_{use} \neq \texttt{@RDM})$$
$$q_{bound} = \texttt{@Immutable} \Rightarrow (q_{use} \neq \texttt{@Mutable} \land q_{use} \neq \texttt{@RDM})$$

The meaning is quite straight forward: if the qualifier is `@Mutable` or `@Immutable`, the valid use is itself or `@Readonly`, otherwise any use is fine on this location. The only case in "otherwise" is `@RDM`, and the "any use" does not includes `@Bottom`, which is only allowed on certain uses such as type parameter lower bound or `null`.

While it is logically correct, thus being used in the previous PICO's formalization, this constraint introduces some problems that hinder the development:

- Since the inference is based on Checker Framework Inference, one goal is to just rely on the higher level APIs of inference, including the 5 basic operations of GUT inference: Subtype, adaptation, Equality, Inequality, and Comparable [15]. Such operations are well-developed and tested for inference, with solid encoding basis. Using only these high-level operations will greatly leverage the burden of development and debugging. Without using them which provided an unified entry for both typecheck and inference, such as using a low-level implication operator, we have do define the actions of PICO in typecheck and inference separately.
- The rule takes two constraints in encoding, which means extending this rule requires making changes on both constraint. For example, if we decide to add a precondition to the validity rule, e.g. "only when the use is not a field", this condition should be added on both of the constraints, which is tedious for the developer to maintain and debug.

3.3.2 Definition of Viewpoint Adapted Subtype

Viewpoint Adapted Subtype (VAS) To avoid using low-level constraints as much as possible, we developed new constraints that are equal to the older ones without using implication, which is called viewpoint adapted subtype. As the name indicates, it is a simple but useful operation that combines viewpoint adaptation and subtype:

$$vas(q_l, q_r) \Leftrightarrow q_l \rhd q_r <: q_l$$

VAS is not commutative, so switching q_l and q_r yields different results. In the validity rule of PICO, q_l is the use type qualifier, and q_r is the declaration type qualifier, the instantiation bound. For example, the multiple constraints above could be simplified as:

$$q_{use} \rhd q_{bound} <: q_{use}$$

To validate that VAS is correct *within the syntax of PICO*, a truth table is provided as Table 3.1. Note that `@Bottom` is not a typically use, so it is handled by other rules, thus not listed in the table.

Use (L)	Declaration (R)	L▷R	(L▷R)<:L
@Mutable	@Mutable	@Mutable	⊤
	@Immutable	@Immutable	⊥
	@RDM	@Mutable	⊤
@Immutable	@Mutable	@Mutable	⊥
	@Immutable	@Immutable	⊤
	@RDM	@Immutable	⊥
@RDM	@Mutable	@Mutable	⊥
	@Immutable	@Immutable	⊥
	@RDM	@RDM	⊤
@Readonly	@Mutable	@Mutable	⊤
	@Immutable	@Immutable	⊤
	@RDM	@Readonly	⊤

Table 3.1: Truth table of Viewpoint Adapted Subtype

Viewpoint Adapted Equality (VAE) Viewpoint adapted equality is a derivative of VAS, and its definition is:

$$vae(q_l, q_r) \Leftrightarrow q_l \rhd q_r = q_l$$

Different from VAS, VAE yields false when the use is `@Readonly`. To demonstrate this, a truth table is also provided as Table 3.2.

From the truth table we can see that the only difference compared to VAS is that when q_l = `@Readonly`, the result is always false, while in VAS the result is always true.

VAE is useful when `@Readonly` is not allowed on the use, for example, a constructor invocation inside the new clause, and `@Readonly` is forbidden on the constructor invocation.

3.3.3 Updated Formalization

All updated formalization where implication constraints are replaced with VAS is shown below. Note that for WF-TYPEUSE, this is not the final version, as we further revise the logic in Section 3.6 to support transitive mutable on fields.

A grey highlight box is applied to the added constraints, and the removed constraints are colored with grey.

Use (L)	Declaration (R)	L▷R	(L▷R)=L
`@Mutable`	`@Mutable`	`@Mutable`	$\top$
	`@Immutable`	`@Immutable`	$\bot$
	`@RDM`	`@Mutable`	$\top$
`@Immutable`	`@Mutable`	`@Mutable`	$\bot$
	`@Immutable`	`@Immutable`	$\top$
	`@RDM`	`@Immutable`	$\bot$
`@RDM`	`@Mutable`	`@Mutable`	$\bot$
	`@Immutable`	`@Immutable`	$\bot$
	`@RDM`	`@RDM`	$\top$
`@Readonly`	`@Mutable`	`@Mutable`	$\bot$
	`@Immutable`	`@Immutable`	$\bot$
	`@RDM`	`@Readonly`	$\top$

Table 3.2: Truth table of Viewpoint Adapted Equality

$$\text{T-SUPER}\quad \frac{\begin{array}{c} kd\ in\ C \quad C <: D \quad typeof(constructor(D)) = \overline{k_{p-D}\ q_{p-D}} \to q_{ret-D} \\ typeof(C, kd) = \overline{-\ -} \to q_{ret-C} \\ q_{ret-D} = mutable \Rightarrow q_{ret-C} = mutable \quad q_{ret-D} = immutable \Rightarrow q_{ret-C} = immutable \\ \Gamma(\overline{z}) = \overline{k_z\ q_z} \quad \overline{k_z <: k_{p-D}} \quad \overline{q_z <: q_{ret-C \triangleright q_{p-D}}} \end{array}}{\Gamma_C \vdash \texttt{super}(\overline{z})\ in\ kd}$$

$$\text{WF-CONS}\quad \frac{\begin{array}{c} q_C = bound(C) \\ q_{ret} = mutable \lor q_{ret} = immutable \lor q_{ret} = receverdependentmutable \\ q_C = mutable \Rightarrow q_{ret} = mutable \quad q_C = immutable \Rightarrow q_{ret} = immutable \\ q_{ret} \triangleright q_C = q_{ret} \\ \vdash = (\texttt{this} : underinitialization\ ret, \overline{g} : \overline{k_g\ q_g}, \overline{f} : \overline{k_f\ q_f}) \\ \vdash_C \texttt{super}(\overline{g})\ in\ q_{ret}\ C\ (\overline{t\ C\ g}, \overline{\overline{t\ C\ f}})\{\texttt{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f} \quad \vdash \texttt{this}.\overline{f} = \overline{f} \end{array}}{\vdash_C q_{ret} C(\overline{t\ C\ g}, \overline{t\ C\ f})\{\texttt{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f}\}\ is\ OK}$$

$$\text{WF-METH}\ \frac{\begin{array}{c}\vdash = (this : k_{this}, \overline{p} : \overline{k_p\ q_p}, \overline{y} : \overline{k_{local}\ q_{local}} \quad \vdash \overline{s} \quad \vdash \Gamma(z) <: t_{ret} \quad q_C = bound(C) \\ q_C = mutable \Rightarrow (q_{this} \neq immutable \wedge q_{this} \neq RDM) \\ q_C = immutable \Rightarrow (q_{this} \neq mutable \wedge q_{this} \neq RDM) \\ q_{this} \triangleright q_C <: q_{this} \\ typeof(m_{super}) = k_{this-super}\ q_{this-super}, \overline{k_{p-super}\ q_{p-super}} \rightarrow k_{ret-super}\ q_{ret-super} \\ k_{this-super} <: k_{this} \quad \overline{k_{p-super}\ q_{p-super}} \quad k_{ret} <: k_{ret-super} \\ q_C \triangleright q_{this-super} <: q_{this} \quad \overline{q_C \triangleright q_{p-super} <: q_p} \quad q_{ret} <: q_C \triangleright q_{ret-super}\end{array}}{\vdash_C t_{ret}\ C\ m\ (t_{this}\ C\ \mathtt{this}, \overline{t_p\ C\ p})\ \{\overline{t\ C\ y\ s}; \mathtt{return}\ \ z;\}\ is\ OK}$$

$$\text{WF-EXTEND}\ \frac{\begin{array}{c}q_D = mutable \Rightarrow q_C = mutable \quad q_D = immutable \Rightarrow q_C = immutable \\ q_C = bound(C) \quad q_D = bound(D) \quad q_C \triangleright q_D = q_C\end{array}}{\vdash C <: D\ is\ OK}$$

$$\text{WF-TYPEUSE}\ \frac{\begin{array}{c}q_C = mutable \Rightarrow (q_{use} \neq immutable \wedge q_{use} \neq RDM) \\ q_C = immutable \Rightarrow (q_{use} \neq mutable \wedge q_{use} \neq RDM) \\ \vdash C\ is\ OK \quad q_C = bound(C) \quad q_{use} \triangleright q_C <: q_{use}\end{array}}{\vdash q_{use} C\ is\ OK}$$

3.4 Mutability of Enums

3.4.1 The Enum Type of Java

The Enum type is a special kind of class in Java. Different from a regular class, enums consists a set of predefined constants, and optionally, *fields and methods*. Each enum constant is like an instance of the enum type with a name, so each constant has its own field values which can also be mutated by assignments and invocations of its instance method, through in many cases the fields should not be mutated.

The code example is a shortened version taken from the Java tutorial [5], which demonstrates an enum type with constants, fields, a constructor, a static field, and an instance method.

```
1 public enum Planet {
2     MERCURY (3.303e+23, 2.4397e6),
```

```
3        NEPTUNE (1.024e+26, 2.4746e7);
4
5        private final double mass;   // in kilograms
6        private final double radius; // in meters
7        Planet(double mass, double radius) {
8            this.mass = mass;
9            this.radius = radius;
10       }
11
12       public static final double G = 6.67300E-11;
13       double surfaceGravity() {
14           return G * mass / (radius * radius);
15       }
16   }
```

3.4.2 Revised Mutability Constraints for Enums

Previously PICO applied implicit `@Immutable` to all enum types. Different from the defaulted `@Immutable`, the implicit `@Immutable` is not overridable by the users, just like the primitives. So, for all enums with fields, their instantiation bounds are locked to `@Immutable`. As a result, all of the use of enum constants are also limited to `@Immutable` and `@Readonly`.

Since there is a possibility that the user requires the field of an enum constant to mutate, it is unreasonable to lock the type of enum to `@Immutable`. So we removed enum from the list of implicit immutable types, but we still default the enum type as `@Immutable`, making it possible for the users to declare a `@Mutable` enum.

After the change, the implicit `@Immutable` object types in PICO are:

- All literals: primitive, string, and class literals. Primitive literals include: int, short, long, float, double, byte and char.
- Reference types: `String`, `Double`, `Boolean`, `Byte`, `Character`, `Float`, `Integer`, `Long`, `Short`, `Number`, `BigDecimal`, `BigInteger`

3.5 Class Extends and Implements

The extends and implements clause are special use of a type. As the same validity rules apply on the use itself, except from `@Readonly` is not usable on a extends or implements clause.

In the changes to PICO, we changed the way PICO interpret the extends and implements clause to make the errors more clear. The process is shown below:

1. If no explicit qualifier presents on the extends and implements clause, apply the default as the instantiation bound of the *extending or implementing class*.
2. Check if the the qualifier on the clause is a valid use of the the clause's type. Additionally, `@Readonly` is not allowed on the use.
3. Compare if the instantiation bound of the extending or implementing class is a valid use of the type of the clause.

In a nutshell, a class can extend a class of the same instantiation bound, or an `@RDM` class. But we further divided the error types to make it more clear to user.

This code example shows the different uses:

```
1  @Immutable
2  class Extender1 extends MutableExtendee {}  // invalid use
3  @Immutable
4  class Extender2 extends @Immutable MutableExtendee {} // invalid use
5  @Immutable
6  class Extender3 extends @Mutable MutableExtendee {}  // invalid extends
7  @Immutable
8  class Extender4 extends RDMExtendee {}  // valid
9  @Immutable
10 class Extender5 extends @Mutable RDMExtendee {}  // invalid extends
```

All of the `Extenders` are `@Immutable` class. For `Extender1`, the extends operation is valid, because the extends clause is equivalent to `@Immutable MutableExtendee`, and obviously, a `@Immutable` class can extend a `@Immutable` class. However, the use of the

type, `@Immutable MutableExtendee`, is invalid, because `@Immutable` is not a valid use of class `MutableExtendee` with a `@Mutable` instantiation bound.

`Extender2` is the same with `Extender1`, only with the `@Immutable` qualifier explicit.

The extend operation on `Extender3` is invalid for the reason that `@Mutable` is not a valid use of `@Immutable`, but the use of class `MutableExtendee` is valid.

For `Extender4`, because the defaulted extends clause `@Immutable RDMExtendee` is a valid use of `RDMExtendee`, and the extend operation itself is also valid, the class is valid.

For `Extender5`, it is also a valid type use if the `RDMExtendee` is used as a `@Mutable` type, but the extend operation is invalid, for a `@Immutable` class cannot extends `@Mutable`.

3.6 Transitive Mutable Field

3.6.1 The Transitive Problem of Mutable Field

For the previous PICO, the fields with `@Mutable` qualifier will always be excluded from the abstract state, which will cause problems for declaring a field:

```
1  @Mutable
2  class A {
3      /*@Mutable*/ B b;
4  }
5
6  @Mutable
7  class B {
8      int field = 0;
9  }
10
11 class Main {
12     void foo(@Readonly A a) {
13         a.b.field = 1; // error?
14     }
15 }
```

In the code example, the field `B b` will be defaulted to `@Mutable` in the previous PICO, and when accessing the field from object `@Readonly A a`, `a.b` surprisingly yields a `@Mutable B`

type. As we stated, `@Mutable` means that the target is removed explicitly from the abstract state, so the usage is not wrong. To make the field not writable, the user can also choose to apply `@Readonly` on the field `b`, but the field cannot be mutated in all instantiations even not referenced by `@Readonly`.

Since the type with qualifier `@Mutable` is always excluded from the abstract state, such use cannot be forbidden, which may leak the field via `@Readonly` references. To solve this, there should be a way for the field with `@Mutable` instantiation bound optionally a part of the abstract state.

Based on the discussion, PICO needs a mechanism to make the field writable via a `@Mutable` receiver, or not writable via a `@Readonly` receiver to ensure flexibility and ease to use. And this should be the default action for such a field.

3.6.2 Expanding the Use of RDM on Field

From the problem, we can see that the field with the instantiation bound of `@Mutable` should have a third option to work differently from `@Mutable` or `@Readonly`. Recall the requirement: the `@Mutable` field should depend on its receiver to determine the type of field access. It seems like it is exactly the job for `@RDM`, as it can be resolved to the receiver's type after viewpoint adaptation of a field access type.

The well-formedness rule prevents PICO to apply `@RDM` to a type with an instantiation bound of `@Mutable`, as the usage qualifier must be a super-type of the instantiation bound or the instantiation bound is `@RDM` (recall that `@RDM` classes can have any kind of use except `@Bottom`). To allow usage of `@RDM` on the field if the instantiation bound of it is `@Mutable`, preconditions have to be added to the well-formedness rule before the viewpoint-adapted subtyping check, to make an exception for field uses. Note here the WF-FLD is not related to the problem. WF-FLD is a set of extra *restrictions* for fields, thus cannot be used to make *exceptions*.

The code example below demonstrates such use:

```
1  @Mutable
2  class A {
3      /*@ReceiverDependentMutable*/ B b;
4  }
5
6  @Mutable
```

```
class B {
    int field = 0;
}

class Main {
    void foo(@Readonly A a) {
        a.b.field = 1; // error!
    }
}
```

By allowing to use `@RDM` on the field `A::b`, the viewpoint adaptation result for the field access on line 13, `a.b`, resolves to `@Readonly A`, preventing the assignment to the `field`.

Only including whether the use is a field is not enough for the precondition, and the first problem is the enclosing class. Not all enclosing classes should be allowed to have such use of the field, and the problem arises when `@RDM` and `@Immutable` become the enclosing class of the field. For example:

```
@Mutable
class MutableBox {
    int field;
}
@Immutable
class IClass {
    @ReceiverDependentMutable MutableBox box = new MutableBox();
}
class Test {
    void foo(@Readonly IClass ri, /*@Immutable*/ IClass ii) {
        ri.box.field = 1; // error
        ii.box.field = 2; // bad type
    }
}
```

The first field access is fine: `ri.box` resolves to `@Readonly MutableBox`, so the assignment to `field` is invalid, getting prevented by PICO. But the second field access has a problem: `ii.box` resolves to `@Immutable MutableBox`, and the type itself is invalid, as `@Immutable` is not a valid use of a `@Mutable` class. Since the receiver of `field` is invalid, the check on the

assignment operation is skipped. To prevent the potentially bad behavior from happening, the use of `@RDM` on the field with `@Mutable` bound within a `@Immutable` enclosing class should be forbidden.

Here is a similar scenario for `@RDM` class:

```
1  @Mutable
2  class MutableBox {
3      int field;
4  }
5  @ReceiverDependentMutable
6  class RDMClass {
7      /*@ReceiverDependentMutable*/ MutableBox box = new MutableBox();
8  }
9  class Test {
10     void foo(@Immutable RDMClass ii) {
11         ii.box.field = 2; // bad type
12     }
13 }
```

For the same reason with the `@Immutable` class example, `ii.box` resolves to a invalid use for `MutableBox`, so the use should be also forbidden on `@RDM` enclosing classes.

The second problem is the instantiation bound of the field itself. For `@RDM`, undoubtedly the usage qualifier on the field could be `@RDM`. And for `@Mutable`, we need that to follow the receiver's qualifier if the receiver is `@Readonly`. But whether allow such use on the field with an instantiation bound of `@Immutable` is a problem. For succinctness of the rules, we are making a safe option that not allowing `@RDM` to be used on a field with the bound of `@Immutable`. The reason is that an object of `@Immutable` class is not mutable after all, regardless of being a field. Depend on the receiver to resolve to `@Readonly` is not particularly useful in this scenario.

Definition Based on the discussion, `@RDM` should also be usable on types in addition to the original well-formedness rules when:

- the type is of a field.
- the instantiation bound of the type is `@Immutable`

- the instantiation bound of the enclosing class is `@Immutable`

To combine this with the well-formedness rule for type use, we have WF-TYPEUSE rule further updated to Fig 3.1.

$$\text{WF-TYPEUSE}\ \frac{\begin{array}{c}\vdash C\ is\ OK \\ q_C = bound(C) \\ q_E = enclosingBound(C) \\ isField(C) \land q_C = \texttt{@Mutable} \land q_E = \texttt{@Mutable} \Rightarrow q_{use} \neq \texttt{@Immutable} \\ \neg isField(C) \Rightarrow q_{use} \triangleright q_C <: q_{use} \\ isField(C) \land q_E \neq \texttt{@Mutable} \Rightarrow q_{use} \triangleright q_C <: q_{use} \\ isField(C) \land q_E = \texttt{@Mutable} \land q_C \neq \texttt{@Mutable} \Rightarrow q_{use} \triangleright q_C <: q_{use}\end{array}}{\vdash q_{use} C\ is\ OK}$$

Figure 3.1: WF-TYPEUSE's second update, adding the extended-RDM rule

Note that *isField* is a helper function added to PICO that returns whether a type is used as a field type. In Java compiler AST, the information is stored in the element. *enclosingBound* is another helper function added to PICO that returns the instantiation bound of the enclosing class, which can be done easily by traversing the father nodes of the field without resulting into another constraint in implementation.

For all possible outcomes for uses of the field in different conditions, please refer to Appendix A for a table of results.

3.6.3 Notable Examples

Inline Initialization This code example demonstrates how `@RDM` field works with inline initialization.

```
1  @ReceiverDependentMutable
2  class RDMBox {...}
3
4  @Mutable
5  class MutableClass {
6      /* @ReceiverDependentMutable */ MutableBox tmbox = new MutableBox();
7      /* @ReceiverDependentMutable */ RDMBox tmrbox = new @Mutable
   ↪  RDMBox();
```

```
8       @Mutable RDMBox nmrbox = new @Mutable RDMBox();  // non-transitive
9
10      ImmutableBox ibox = new ImmutableBox();
11      @Immutable RDMBox irbox = new @Immutable RDMBox();  // need explicit
        ↪   annotation
12  }
```

At first glance, assigning a `@Mutable` type to `@RDM` type is an invalid assignment. Because the two qualifiers are at the same level of the hierarchy, such assignment is neither upcasting nor downcasting. But note that the assignment happens on a field, so it is considered a part of the initialization process, as in a constructor. Checker Framework will assume such assignment happens in the default constructor, so the assignment equals to `this.tmbox = new MutableBox();` inside a constructor where the receiver `this`'s type is `@Mutable MutableClass`. After viewpoint adaptation, the type of field resolves to `@Mutable MutableBox`, such assignment is valid.

RDM Defaulting in RDM Class `@RDM` is not allowed in fields with instantiation bound of `@Mutable` in `@RDM` class, but PICO will still default the field to `@RDM` even it is illegal. This will raise an error if no annotation is provided, forcing the programmer to provide an explicit annotation for the field to avoid careless errors:

```
1   @ReceiverDependentMutable
2   class RDMClass {
3       /* @ReceiverDependentMutable */ MutableBox mbox1; // error will
        ↪   raise
4       @Mutable MutableBox mbox2;
5       @Readonly MutableBox mbox3;
6   }
```

In inference, since such use is forbidden by the formalization, the qualifier on the field with instantiation bound of `@Mutable` in a `@RDM` class will never be inferred to `@RDM`.

3.6.4 Alternative Solutions

Besides the solution we finally used, we also explored several alternatives that actually could solve the issue we presented in this section, but not a valid or ideal solution. Here we present the alternatives to show how our final solution is developed.

Make RDM Default on Class Declaration At first glance, the transitive field problem can be easily resolved just by changing the defaulting rule of class instantiation bound to `@RDM`. So the code example becomes:

```
@Mutable
class A {
    /* @ReceiverDependentMutable */ B b;
}

/* @ReceiverDependentMutable */
class B {
    int field = 0;
}

class Main {
    void foo(@Readonly A a) {
        a.b.field = 1; // error!
    }
}
```

By using `@RDM` as the default, the field `b` can use `@RDM` as the declaration qualifier, thus gain the transitive mutability by a receiver-sensitive qualifier. In the example, the field access `a.b` yields a type of `@Readonly B`, effectively preventing the mutation on the field by assigning to the `b.field`.

The updated rule will work for the example provided and will work for unannotated code as well. But the solution still has a problem: if class `B` is explicitly declared as `@Mutable`, just like class `A`, the field immediately loses the ability to be transitively mutable. So, this solution does not meet all requirements of the problem.

Making RDM the Supertype Similar to the solution we discussed in 3.6.2, if we want to allow receiver-transitive on a field, providing a way to use `@RDM` on the `@Mutable` type is an easy way. Another possible solution is changing the hierarchy of the mutability qualifiers, making `@RDM` the supertype of `@Mutable`, thus allowing objects with instantiation bound of `@Mutable` to be assigned to a `@RDM` reference, known as upcasting. The qualifier hierarchy after adaptation is showed in Figure 3.2.

Note that `@Immutable` remained its position in the hierarchy, not also being the subtype of `@RDM`. One reason is that the `@Immutable` qualifier does not need a similar feature as

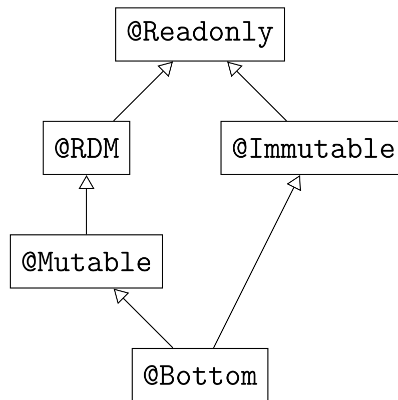

Figure 3.2: Hierarchy of qualifiers of RDM being supertype

`@Mutable`, because their objects cannot be modified in any way after initialization, regardless of having `@Readonly` on the use. So, being receiver-sensitive does not have benefits in this case. But a more serious reason is that allowing such hierarchy will potentially break the immutable guarantee of `@Immutable`. Because a `@RDM` field may be viewpoint adapted to `@Mutable`, if the underlying object is `@Immutable`, it can be mutated via the viewpoint-adapted `@Mutable` reference. To demonstrate the problem, a code example is provided below:

```
1  @Immutable
2  ImmutableBox {
3      int immutableField = 0;
4  }
5
6  @ReceiverDependentMutable
7  class RDMBox {
8      // the assignment is allowed by the hierarchy
9      @ReceiverDependentMutable ImmutableBox boxField = new ImmutableBox();
10 }
11
12 class Test {
13     void foo(@Mutable RDMBox box) {
14         box.boxField.immutableField = 1; // error
15     }
16 }
```

As shown in the code example, if `@Immutable` is assigned as the subtype of `@RDM` in the hierarchy, the `@RDM` will be permitted on the field of `boxField`, and the assignment of `@Immutable` object is also permitted as upcasting. But in the use of the class `@Mutable RDMBox box` which is allowed as a `@Mutable` variation of `RDMBox`, `box.boxField` is resolved to `@Mutable`, exposing a `@Mutable` reference to an object with `@Immutable` instantiation bound, breaking the guarantee that an object of `@Immutable` class can never be mutated after initialization. As a result, this hierarchy is not safe for PICO.

But with the change to the hierarchy alone is not enough. Similar to the example ahead, the hierarchy of this approach may still cause dangerous operations. For example:

```
@Mutable
MutableBox {
    int mutableField;
}

@ReceiverDependentMutable
class RDMBox {
    // the assignment is allowed by the hierarchy
    @ReceiverDependentMutable MutableBox boxField = new MutableBox();
}

class Test {
    void setSomeImmutable(@Immutable Object o) {...}

    void foo(@Immutable RDMBox box) {
        setSomeImmutable(box.boxField); // error?
    }
}
```

The code example is a slightly altered version. The difference is that now the field of `RDMBox` is a reference of `@Mutable` class. In this case, the type of `box.boxField` is `@Immutable MutableBox`, having a contradiction in the type. Later on, if this type is needed to pass into some methods that require an `@Immutable` parameter, like `setSomeImmutable`, this dangerous operation is not uncovered by PICO. The reason for such operation being dangerous is that there may be `@Mutable` alias of the underlying object `box.boxField`, which is absolutely legal because the class instantiation bound of `MutableBox` is `@Mutable`. When `setSomeImmutable` is expecting a `@Immutable` object and assumed that its content can

never change, the object may be mutated via outer alias, breaking the assumption of `setSomeImmutable`.

To ensure the assignments do not violate the immutable guarantee, some limits have to be added to this solution. For example, only the field can upcast `@Mutable` to `@RDM`. Since the limit should be made on the only use, it is confusing to the user why PICO has such a hierarchy of qualifiers. So, this solution is unfriendly to the users, and because of the number of limits it has to add, it is hard to implement.

Introducing Qualifier for Transitive Mutable Making `@RDM` directly the supertype of `@Mutable` will introduce a problem on non-field uses or having strange viewpoint adaptation results. To address the problem by preserving the current typing rules, a new qualifier `@TransitiveMutable` can be introduced to PICO to add new logic on fields with a `@Mutable` instantiation bound. The new qualifier `@TransitiveMutable` is added as the supertype of `@Mutable`, and the hierarchy is shown in Figure 3.3.

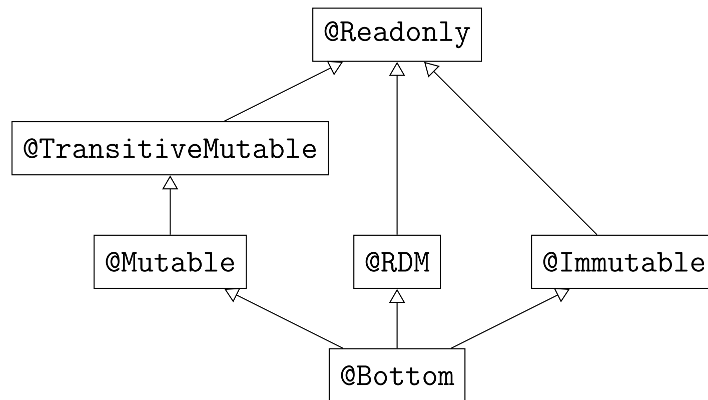

Figure 3.3: Hierarchy of qualifiers including `@TransitiveMutable`

The newly added qualifier, `@TransitiveMutable` acts like a `@RDM` with less viewpoint adaptation rules to ensure it will not be resolved to invalid types like `@Immutable MutableBox`. To ensure this not happen, the following viewpoint adaptation rules are added to PICO:

$$\texttt{@Mutable} \triangleright \texttt{@TransitiveMutable} = \texttt{@Mutable}$$

$$\texttt{@Readonly} \triangleright \texttt{@TransitiveMutable} = \texttt{@Readonly}$$

$$\texttt{@Immutable} \triangleright \texttt{@TransitiveMutable} = \texttt{@Mutable}$$

$$\texttt{@RDM} \triangleright \texttt{@TransitiveMutable} = \texttt{@Mutable}$$

$$\texttt{@TransitiveMutable} \triangleright \texttt{@TransitiveMutable} = \texttt{@TransitiveMutable}$$

The difference of the viewpoint adaptation rules compared to the normal `@RDM` is that the `@TransitiveMutable` preserves the `@Mutable` if the receiver is a type other than `@Mutable`. But the qualifier also works just like `@Mutable` when checking the mutability of a type, allowing the mutation to the type. The following code example demonstrates this feature:

```
@Mutable
MutableBox {
    int mutableField;
}

@ReceiverDependentMutable
class RDMBox {
    @TransitiveMutable MutableBox boxField = new MutableBox();
}

class Test {
    void setSomeImmutable(@Immutable Object o) {...}

    void foo(@Immutable RDMBox box) {
        box.boxField.mutableField = 1;
        setSomeImmutable(box.boxField); // error
    }
}
```

From the code, we can see that although the receiver of the field access `box.boxField` is `@Immutable`, the type after viewpoint adaptation remains `@Mutable`. Recall that by using the `@Mutable` on the class, we mean that the instances of the class are always mutable, so the assignment on line 15 is not an expected error. Adding a qualifier is not enough, to ensure the mutations are checked correctly, we have to update the T-FLDASS rule in PICO to ensure the newly added qualifier is treated as `@Mutable` in assignments, as shown in Fig 3.4.

Another limit is that `@TransitiveMutable` cannot be used as the instantiation bound, because using that as instantiation bound is not meaningful, as it acts the same with

$$\text{T-FLDASS}\ \frac{\begin{array}{c}\Gamma(x) = k_x\ q_x\ C \\ \Gamma(y) = k_y\ q_y \\ fType(C, f) = a_f\ q_f \\ q_x = \texttt{@Mutable} \\ \vee q_x = \texttt{@TransitiveMutable} \\ \vee(k_x = \texttt{@UnderInitialization} \wedge q_x = \texttt{@Immutable}) \\ \vee(k_x = \texttt{@UnderInitialization} \wedge q_x = \texttt{@RDM}) \\ \vee(a_f = \texttt{@Assignable} \wedge (q_x \neq \texttt{@Readonly} \wedge q_f \neq \texttt{@RDM})) \\ q_y <: q_x \triangleright q_f \\ k_x = \texttt{@UnderInitialization} \wedge k_y = \texttt{@Initialized}\end{array}}{\Gamma \vdash x.f = y}$$

Figure 3.4: The formalization for field access with `@TransitiveMutable`

`@Mutable`. Since WF-CLASS only allow 3 qualifiers, `@Mutable`, `@Immutable` and `@RDM` on the class declaration, there is no need to update the rules.

Also, since the static field cannot have a receiver, the receiver sensitive qualifier is not meaningful. So, `@TransitiveMutable` should not be used on a static field, thus the WF-STATIC-FIELD rule should be updated:

$$\text{WF-STATIC-FIELD}\ \frac{\begin{array}{c}fType(sfd) = a\ q \\ q \neq \texttt{@TransitiveMutable} \\ q \neq \texttt{@RDM} \\ q \neq \texttt{@PolyMutable} \\ a \neq \texttt{@RDA}\end{array}}{\vdash_C sfd\ is\ OK}$$

Figure 3.5: The formalization for static field with `@TransitiveMutable`

Finally, to ensure is this the default behavior for a field with `@Mutable` instantiation bound, the defaulting rule needs to be updated. For field, if the instantiation bound of the field is `@Mutable`, the defaulted qualifier should be `@TransitiveMutable`. Otherwise, the defaulted qualifier should be its instantiation bound.

This is a working solution that passes all the test cases in PICO. And compared to the other alternative working solution, making `@RDM` the supertype, the introduced rules are easier to implement. The reason that we did not choose this one is introducing a new qualifier will make the system too complicated. To reduce the rules that the user needs

to learn, we finally used the solution presented in 3.6.2, expanding the use of @RDM on the fields.

3.7 Configurable Deep Immutability

3.7.1 Problem of Strict Deep Immutability

Deep immutability is enforced by many immutable checkers and it guarantees that all of the fields are recursively immutable, so the whole object remains the same.

A basic requirement for deep immutable is that all the fields are final (not re-assignable) and immutable. If a field is mutable, the deep immutability guarantee is broken and parts of the object may be mutated.

However, in some cases, when the mutability type of a certain field needs to be intentionally excluded from the deep immutability while other fields not, the user is often not provided a way to exclude that field, and have to declare the whole class as mutable or suppressing warnings.

For example, a class with two key fields is used as the key of a HashMap, and the key fields are properly @Immutable, but the class has a field of lastAccessed for statistic purposes. While the hashCode and equals is implemented only with the two key fields, it should be safe to use as key even the lastAccessed field is mutable. Declaring the class to be @Mutable is feasible when the checker reports any @Mutable object is used as the key of HashMap. So, a mechanism should be provided to the power users to disable deep immutable checks on certain fields when they are confident that they are not cared for by the system.

```
1  @Immutable
2  class NeedShallowImmutable {
3      ImmutableBox key1;
4      ImmutableBox key2;
5
6      @Override
7      int hashCode() {
8          // only key1 and key2 requires transitive immutable
9          return Objects.hash(key1, key2);
10     }
```

```
11      @Override
12      boolean equals(Object other){
13          return other instanceof NeedShallowImmutable &&
         ↪  ((NeedShallowImmutable) other).key1.equals(key1) &&
         ↪  ((NeedShallowImmutable) other).key2.equals(key2);
14      }
15
16      @Mutable Date lastAccessed; // not cared
17  }
```

Some checking tools support extra qualifier to allow immutable container having contents of mutable types, such as the `@Immutable(containerOf = ...)` method in Google's Error Prone [3], where user can specify a type variable on the class to exclude the field of the type from the deep immutability scope. This proved useful when a container only needs deep immutability on its non-content fields, such as the `head` or `size` fields in a list.

Although this approach is much more flexible than the tools that enforce deep immutability, one limit still prevents it to be further flexible: the target of `containerOf` must be a type variable. That means a container without a type parameter cannot exclude a content field from deep immutability. For example:

```
1  @Immutable
2  class ShallowContainerWithoutGeneric {
3      @Mutable MutableClass content;
4
5      // Below is the container internal fields which needs transitive
       ↪  immutable
6      Object head;
7      Object tail;
8      // ...
9  }
```

One real-world example of such use is the container of the API response. Often such responses only have certain kinds of the header, and these headers are cached in a pool. When using them, only the payload is replaced with the data, and the header remains the same. In this case, only the header fields require deep immutability. Since the payload is often just a char array, no type parameter is needed on the wrapper class.

3.7.2 Enjoy Flexibility with Configurable Deep Immutability

Different from many tools that enforce deep immutability without an option to disable or limit to type variables, PICO enforces deep immutability by default but provides an option to explicitly exclude a field from deep immutability, allowing shallow immutability on certain fields inside a `@Immutable` class without warnings.

PICO uses a syntax that does not requires additional qualifiers or parameters, reducing the learning cost of the users.

1. When a field has an explicit `@Mutable` qualifier, it means that the user explicitly exclude the field from the deep immutability guarantee, leaving the responsibility of the field to the user.
2. When a field does not have an explicit `@Mutable` qualifier, it means that PICO should enforce deep immutability on this field, and issue a warning when the instantiation bound of the field is `@Mutable`.

It worth note that for a `@Mutable` use of a `@RDM` class inside a `@Immutable` class, since an explicit qualifier is required, it is regarded as an explicit use of `@Mutable`, so PICO does not issue warnings on this.

The code example below demonstrates the configurable deep immutable rules in PICO.

```
1  @Mutable
2  class MutableBox {...}
3  @Immutable
4  class ImmutableClass {
5      // this implicit use of mutable will issue a warning
6      MutableBox implicit = new MutableBox();  // warning
7      // this explicit use of mutable will pass the check
8      @Mutable MutableBox explicit = new MutableBox();
9  }
```

From the discussion and the examples, we can conclude that compared to Glacier, PICO further supports selective shallow immutable to enhance flexibility. And compared with Error Prone, PICO can apply the exemption to any field instead of limiting the use to type variables. Besides, deep immutability is still enforced by default to prevent human errors in the code.

3.8 Instance Methods for Array

During the experiments, we found that PICO has issues with arrays' instance methods: all the method invocation of arrays, such as array's method `clone` and inherited method `hashCode`, are defaulted like a fully unannotated code with the receiver type of `@Mutable`, preventing the method be invoked on `@Immutable` arrays.

```
1 void foo(@Immutable Object @Immutable [] arr) {
2     @Immutable Object @Immutable [] cpy = arr.clone();  // false
      ↪  positive
3 }
```

In the code above, when checking the invocation of `arr.clone()`, the receiver type will be defaulted to `@Immutable Object @Mutable []`, preventing `arr` invoking the method. However, since the return type will be fixed by Checker Framework, no error is issued on the assignment.

This issue is related to the Checker Framework's procedure for code without source files. When a source code is unavailable for the checker, for example, Java built-in and library classes, Checker Framework will first try to find that in the stub files where the developer can provide an annotated signature. If a signature is found, Checker Framework will use that as if it is from the class declaration.

Although arrays are treated like objects, there is no class declarations of them in Java runtime, and its creation is not a typical `new` operation, but implemented by special JVM instructions: `newarray` for primitive arrays, `anewarray` for object arrays and `multinewarray` for multi-dimensional arrays [6]. The implementation details of the array are decided by the JVM, but not included in the Java runtime [17]. Since array objects are not associated with any classes, stub files do not affect the instance methods of arrays, and the declaration types for the instance methods have defaulted like unannotated code.

To solve the problem, PICO uses an annotator during the generation of method declaration signatures before the defaults are applied. Because the programmer cannot override or extend the array type, it is guaranteed that the instance methods are fixed: `clone` and all methods inherited from `java.lang.Object`. So the annotator firstly checks whether the receiver of the method is an array, then applies qualifiers to the receiver type. Since all methods of an array are invocable on any mutability type of receiver, PICO will assign `@Readonly` to the receiver.

PICO does not need to deal with the return type of `clone`, because the Checker Framework will replace the return type with the receiver's type on array's `clone` method invocation, ensuring the cloned type is the same with the receiver. So it is safe to leave the return type of `clone` untouched in the annotator.

3.9 Casting

PICO was designed to issue warnings on the casts that are possibly unsafe. For example, PICO would issue a warning on the casting in the code below, which is removed in our improvements.

```
1 int compare(Object o1, Object o2) {
2     Integer i1 = (@Immutable Integer) o1; // no warning
3     // ...
4 }
```

If the Java compiler is convinced that the type in the casting operator is correct, then casting the qualifier to the bound has no problem. For example, for reference `@Readonly Object` casting to `Integer`, if the compiler issues no warnings on that, then casting to `Integer` should have no problem, and casting to the bound, i.e. `@Immutable Integer`, should also have no problem.

However, if the cast itself is invalid, e.g. no subtyping relations between the two classes, the Java compiler will stop compiling and the code cannot reach the annotation processor, PICO, which is out of the scope of PICO's check.

3.10 Polymorphic Qualifier in Inference

It is hard to infer that a location can potentially use a polymorphic qualifier, so PICO is not implementing this. However, the existing `@PolyMutable` can be correctly substituted during both the inference or the typecheck mode of the inference checker.

Previously PICO does not support `@PolyMutable` in the inference checker, so during the typecheck phase of the code inference, the inference checker will stop working when encounters the `@PolyMutable` qualifier, complaining about an unsupported qualifier in the code.

To fix this issue, we added `@PolyMutable` to the supported qualifiers to the inference checker. To ensure this qualifier does not appears in the inference result, we added inequality constraints on all type use locations during inference only, forcing all use locations not to infer to `@PolyMutable`, ensuring no new polymorphic qualifiers get inferred, but only reusing the existing `@PolyMutable` as constant variables. By doing this, the polymorphic qualifiers are supported during the two typecheck phases of the round-trip inference.

3.11 Static Inner Classes

PICO does not allow the use of `@RDM` in static context. This makes sense because a static block, method or field cannot have receiver, so the qualifier cannot resolve to anything.

However an exception are the static inner classes. Being no different than a regular class in the sense of using `@RDM`, the use of `@RDM` is still be allowed on the declaration of a static inner class.

```
1  @Immutable
2  class Outer {
3      @ReceiverDependantMutable  // allowed
4      static class Inner {...}
5  }
```

3.12 Action after Error

Previously PICO will stop processing a part of the code if an error is raised. Undoubtedly this will improve the performance, but may also frustrate a user if more error appeared after each edit.

Another issue is that the inference checker and the typecheck checker have a little inconsistency in dealing with the error. For some errors, typecheck checker would stop processing the code, while the inference checker still moves on. Also, during automated testing, it is better to have all expected errors of a test case instead of consuming a part of it.

To unify the action between the checkers and enhance the testing experience, we removed the logic that prematurely stops the checking if an error happens, forcing the checker

to complete the check of compound code, such as inline initialization. The code example below shows the difference.

```
class MyClass {
    // Field type error, constructor invocation OK.
    @Immutable MutableClass field1 = new @Mutable MutableClass();
    // Field type OK, constructor invocation error.
    @Mutable MutableClass field2 = new @Immutable MutableClass();
    // Both of the use is error. Raise two errors for the problem.
    @Immutable MutableClass field3 = new @Immutable MutableClass();
}
```

Previously, PICO stops checking `field3` after issuing the error that the type use is wrong. After the change, PICO will raise two errors for the type use and constructor invocation respectively.

To summarize, this chapter introduced the updates to PICO to enhance flexibility and improved some bad designs.

- The class bound is clarified firstly, with the refined syntax for bound of anonymous classes. The logic is also updated for Enum classes and the class extends and implements clauses to ensure the class bound is enforced correctly.
- The ordering of polymorphic substitution, viewpoint adaption, and type variable resolving is updated to avoid introducing un-substitutable qualifiers to polymorphic substitution.
- To fix the deep immutability/read-only problem, different fixes are proposed for mutable and immutable classes.
- Viewpoint adapted subtype is introduced to provide syntactic sugar for the formalization of `@RDM` classes.
- Minor bug fixes, including the arrays' instance methods, casting logic, using polymorphic qualifier during inference, updating the static scope, and unified the action after encountering an error.

Chapter 4

Experiments

This chapter introduces the experiments with PICO. To reproduce the experiments, a docker image of PICO is provided: https://hub.docker.com/repository/docker/lnsun/pico.

4.1 Continuous Integration Tests

Automated unit testing is included in the continuous integration to detect some kind of errors in implementation. Based on the existing test cases of PICO, we added more test cases to further enhance the correctness of the result. The added test cases includes type-checking the inference test cases, and a subset of Glacier test cases [11].

Sharing Test Cases between Typecheck and Inference In the previous version of PICO, the typecheck and inference checker use a different set of test cases for automated testing. Since the type rules of typecheck part and the inference part are the same, the test cases should be shareable between the two checkers.

It is quite simple to use the test case of inference for type check, but it does not applies vice versa. The inference tool does not supports initialization sub-checker for some technical reason of the inference framework. Besides, the inference tool cannot infer new `@Assignable` and `@PolyMutable`, although the existing qualifiers can be used. Based on the reason above, the test cases of the type check cannot be reused to inference directly and requires adaptation. Since there is no support for ignoring certain errors on the checker

level, we are not reusing the type check's test cases for inference, but only reusing the inference's test cases for type check.

The test cases added to the type check revealed some inconsistency of the two implementation of PICO:

- Actions on error (see 3.12): the inference tool will continue checking when the clause of the code have raised error, but typecheck will skip the tree when an error arises. We unified the behavior to expose more error to the users and not try to be too "clever".
- Inconsistent error keys: Inference and typecheck have different error keys on the same error, e.g. an invalid extends clause. This issue is not hard to solve, just unify the error key strings in configuration.
- Extra checks: Inference checker will also checks each super class up to `Object` when checking each class tree, while typecheck checker does not have such behavior. After carefully inspection of the formalization, we decided that the check is not necessary in inference, so this check and corresponding error key is removed from the inference checker.

4.2 Comparison with Glacier

Glacier is a similar checker for immutability [12], and its qualifier is partly compatible with PICO, so we are reusing a compatible subset of the test cases of Glacier to test scenarios that we did not cover.

Although some test cases can be just imported to PICO and use, many of them still require changes to work with PICO. One reason is the qualifiers: Although Checker Framework supports aliasing for the qualifiers, if the used qualifier is not included in the path, the Java compiler will raise an error and cannot proceed to the annotation processor. We are not including Glacier in our build since it relies on an older version of Checker Framework, and it does not comply to the Checker Framework version we are using in PICO. As a result, we have to manually replace the import clause with our qualifier.

Another reason are the error keys. Glacier uses a different set of error keys compared with PICO and it is not purely a mapping from A to B. The difference in processing the code result into different errors. For example, for an invalid case of implementation where annotation is absent on the implements clause, PICO first defaults the clause to the

same qualifier of the class bound, so the implementation is valid. But since the defaulted qualifier is invalid on the implemented type, there will be an error of invalid annotation on the use. But for Glacier, since it only supports class-level, there is no "use annotation" on the type. So when that is invalid, Glacier will just inform the user that the implement clause is invalid.

Different defaulting scheme also contributes to some of the incompatible tests. For example, the `Object` in PICO have a bound of `@RDM`, so the default use type of it in return type will be `@Mutable` to be compatible with most uses. In the Glacier test cases where `@Immutable Object` are returned, the return type of the method can be unannotated. This design difference requires such test case to be annotated explicitly.

4.2.1 Improvements to PICO

The PICO adopted many useful changes from Glacier:

Stub File Glacier contains more built-in types in Java that should be `@Immutable`. PICO added such entries to the stub file to better typecheck Java built-in types.

Class Name	Change
java.util.AbstractCollection	add `@Readonly` to the receiver of `size`
java.util.Arrays	add `@Readonly` to the receiver of `copyOf`

Table 4.1: Entries added to stub file

Array Instance Method Some test cases of Glacier unrevealed a serious issue in PICO that the array's invocation methods are not handled correctly. We fixed this problem as discussed in Section 3.8.

Handling of Null For PICO, `null` cannot be other type than `@Bottom`. During inference, a slot is still generated on uses of `null` with the mandatory constraints to ensure it is correctly inferred to `@Bottom`. In order to reduce the amount of the variable slots and constraints, a constant slot of `@Bottom` is applied on the `null` locations, and no constraint is generated if there is no other uses of the slot. This is inspired by how the Glacier handles null, although Glacier does not support inference.

4.2.2 Comparison with Glacier

While we learnt much from Glacier and made updates to PICO, we also noticed some design issues of Glacier, and PICO gives a better solution.

Class Extending Rule There is no `@Mutable` in Glacier. Instead, Glacier have the qualifier `@MaybeMutable` as the super type of `@Immutable`. So, Glacier allows `@Immutable` classes inherit from `@MaybeMutable` classes. In order to prevent aliasing, upcasting between these two types are prohibited.

PICO assign `@Mutable` and `@Immutable` the same level in the hierarchy, so the `@Immutable` class cannot extend the `@Mutable` class. In PICO, `@Mutable` means that the object of the class is *always* mutable. If a common part is going to be shared between `@Mutable` and `@Immutable` class, PICO's solution to this is the `@RDM` class, which will enforce a safe assumption on use, allowing the class can be used both as `@Mutable` or `@Immutable` without unsoundness.

Deep Immutability Glacier enforces deep immutable and this behavior is not overridable, unless the error is suppressed. That means any `@MaybeMutable` fields are not allowed inside an `@Immutable` class. This grunts more safety at the cost of flexibility. If the mutability of a field is not interested in a immutable class, for example, a "last accessed" field of date which is never used in the `hashCode` method, the user cannot override the behaviour.

In PICO, deep immutable is by default but not enforced. As discussed in 3.7, shallow immutability is possible with an explicit `@Mutable` on the field. Otherwise, if no qualifier is provided on the field, deep immutability rule is applied, and issues a warning if the field is a reference to a `@Mutable` class.

Due to this difference, the test cases that tests deep immutability are generally not applicable to PICO if the qualifiers are explicit. PICO is only importing the tests without qualifiers on the field.

Qualifier Position Some Glacier test cases put the qualifier before the modifiers, which will result into a warning. When importing such tests, we changed the order of the modifier before the qualifiers to eliminate the warnings.

Initialization Check In order to reduce human error, PICO used the initialization checker as an sub-checker to get the initialization status of the use of type. In `@Immutable`

class, if one field is not explicitly initialized, both inline initializing or in constructor, PICO will issue an error that the field is not initialized. Because after the constructor, the object can never mutated as guaranteed by PICO, an implicit `null` field seems meaningless in most cases. However, if the user need the field to be null on purpose, an explicit assignment to `null` will disable the error.

While Glacier seems not to have this feature, all uninitialized fields are not expecting an error in the tests. Since this feature is useful, we decided not to disable it, but suppressing the initialization errors in all Glacier test cases.

Type Refinement with Null The PICO enabled the type refinement on local variables. For example:

```
1 class PlainObjects {
2     public void takeImmutableObject(@Immutable Object o) {};
3 
4     void foo() {
5         Object o1 = null;
6         takeImmutableObject(o1);  // error?
7     }
8 }
```

In PICO, the local variable `o1` inside `foo()` is affected by type refinement. After the assignment to `null`, the type will be refined to `@Bottom`, the type for `null` in PICO. So, in the later invocation `takeImmutableObject(o1)`, no errors will be raised, because `@Bottom` types are assignable to any parameter.

However, Glacier seems to not have this feature enabled, thus expecting an error on the method invocation. Since type refinement is useful to reduce the annotation burden, we are removing this expected error from the test when imported to PICO.

The full list of tests imported from Glacier is listed in Table 4.2.

Name	Changes	Problem Found in PICO
ArrayClone		Array's clone method
ArrayParameter	error key, qualifier name,	Added Arrays.copyOf to stub
Arrays		
ClassReturn		
ClassGetName		
CompareTo		
ConflictingAnnotations	qualifier name,	
ConstructorAssignment	error key,	
EnumTest		
FontImpl	error key,	
ImmutableArray	suppress initialization errors	
ImmutableClassMutableInterface	make default explicit	
ImmutableCollection		Add AbstractCollection to stub
ImmutableConstructorInMutableClass	styling	
ImmutableInterfaceClass		
ImmutablePrimitiveContainer	error key, suppress initialization errors	
IncorrectUsage	qualifier name	
IntArgument		
IntReturn		
IntegerAssignment		
InterfaceSupertype		
InvalidImmutableInterfaceClass	error key,	
InvalidTypeArguments	qualifier name	
MethodInvocation		
MethodTypeParameters		
NestedGenerics		
NullAssignment		
Override		
ReadOnlyObject	add explicit qualifier	

Name	Changes	Problem Found in PICO
StringTest	suppress initialization errors	
Transitivity	suppress initialization errors	
Unboxing		
UnmodifiableCollection		
UnmodifiableList		
ValidImmutableInterfaceClass		

Table 4.2: Tests imported from Glacier

4.3 Comparison with ReImInfer

ReIm and ReImInfer is another checker for immutability [21], and we will conduct the comparison by running their inference test cases with PICO. Since all inference test cases in ReImInfer are fully unannotated, there will be no replacing of qualifiers to reuse these test cases.

While PICO inference checker can infer a valid result for most test cases, some of the test cases also revealed a bug in the toolchain of PICO, the Annotation-tools: it cannot insert inferred annotation for array initializer when the enclosing class is not the first class in the file. Since this problem is not within the scope of this thesis, we not not discuss the problem in detail [1].

4.4 Whole Program Inference

For the typecheck checker, dealing with unannotated code will give many false positives, because the typecheck relies on explicit qualifiers to work correctly in many cases. Since the default qualifier applied on the unannotated types are only for the most likely type on a position, there will still be some cases that not possible to be covered by the default.

The whole program inference will go through all uses of a type in the whole project, and based on the existing formalization, the inference checker will infer a possible qualifier for the type and all its use.

Due to technical issues, currently we can only infer one qualifier hierarchy during inference, so assignability and initialization qualifiers cannot be inferred, but the user can still provide explicit qualifiers for the assignability and initialization.

The inference will not infer libraries that are not a part of the project, including Java runtime libraries and other third-party libraries in the dependencies. For the qualifiers of classes and methods from these code, PICO inference will first try to look up the stub file, if their qualifiers are not specified in the stub files, PICO will apply defaults based on the configuration. There are three strategies of assumption: pessimistic, realistic, and optimistic assumptions for libraries:

- Pessimistic assumptions: every library method returns `@Readonly` objects and accepts `@Bottom` arguments. It is supported by the Checker Framework.

[1] For details, please refer: https://github.com/opprop/immutability/issues/22

- Realistic assumptions: every library method returns `@Mutable` objects and accepts `@Mutable` arguments if the type of arguments and returns is not implicit immutable. Implicit immutable will still apply and replaces the `@Mutable` default to `@Immutable`. This this the same defaulting scheme with typecheck and will be used by default.

- Optimistic assumptions: every library method returns `@Bottom` objects and accepts `@Readonly` arguments. Additionally, it is currently a PICO-specific configuration.

Note that the pessimistic assumptions are almost unusable in PICO like for other type systems, because the only `@Bottom` type allowed in user code is null reference and type variable lower bound. Applying pessimistic assumption will make all library method not invocable unless all the arguments are null. Experiments are conducted with both realistic and optimistic configuration.

We experimented the whole program inference on some real-world projects, and analyzed the inferred results, such as the slots and constraints created and the qualifier inferred. The number of the qualifier inferred is shown in Table 4.3. Together with the processing time, the number of variables and slots can be used to estimate the solver's overhead during the checking. The percentage of each kind of qualifier can give us an overview of inferred result. Generally, the inference checker of PICO is usable on real-world projects with acceptable inference results.

	`@Mutable`	`@Immutable`	`@RDM`	`@Readonly`	`@Bottom`
imgscalr	539 (52.9%)	298 (29.2%)	4 (0.4%)	114 (11.1%)	64 (6.2%)
jama	726(25.2%)	1327(46.2%)	15(0.5%)	386(13.4%)	421(14.6%)
react	7160(44.2%)	3120(19.2%)	847(5.2%)	4579(28.2%)	503(3.1%)
exp4j	379 (19.9%)	969 (51.0%)	66 (3.5%)	382 (20.1%)	105 (5.5%)
ECC-RSA-Backdoor	1484 (52.9%)	372 (13.2%)	29 (1.0%)	739 (26.3%)	181 (6.5%)

Table 4.3: Number and percentage of qualifiers inferred by project

A full list of the projects with their repository URL is available in Appendix B.

Chapter 5

Conclusion and Future Work

In this thesis, we presented work that improves the soundness and friendliness of PICO, including updates to the formalization and the defaulting scheme, and experiments on real-world projects and smaller code snippets from other immutability type systems.

We examined and fixed the defaulting scheme and certain bad designs of the original PICO, including the transitive field problem, the ordering of the qualifier and type variable substitution, and the enum classes. For the transitive field problem, we also provided all the approaches we have experimented, with an alternative solution.

We also improved the friendliness of the original PICO, including updating the defaulting scheme to avoid the unsafe defaults, and updated the class and extends clause default to relieve the annotating burden. We also updated the casting behavior and the mechanism of static classes to avoid a false negative warning.

Finally, we conducted the experiments of the improved PICO. We improved the automated testing process of the PICO by adding the cross-check of inference's test cases and adding the Glacier's test case. We also conducted a comparison between PICO and Glacier. We made improvements to PICO based on the Glacier's advantages on library methods and array methods, and we also analyzed the disadvantages of the Glacier where PICO outperforms.

The work in this domain is not yet finished.

Currently PICO lacks the knowledge of third-party libraries by default, and it relies on the user to specify which class should be immutable or mutable. We plan to investigate more open-source projects to add more pre-defined immutable types to the stub file.

The inference result is hard to interpret, especially in whole-program inference when a contradiction in constraints occurs. This can either mean that the project itself has a contradiction with user-defined or pre-defined mutability rules, or it means a bug in PICO. While the result only indicates the slot number and a position in the code, it cannot have more information. Providing a specific reason for the failed inference may benefit the user.

During the inference, when encountered third party libraries, PICO will not infer the qualifiers of their classes and methods, but use the stub file, or configurable assumptions for the default if the stub file does not contain their qualifiers. One possible solution to this problem is to also infer the used method signature to a stub file, and use that to check with the library.

References

[1] Checker Framework Inference: Inference of pluggable types for Java. https://github.com/opprop/checker-framework-inference. Accessed: 2021-03-11.

[2] Dataflow: enhancing flow-sensitive type refinement. https://checkerframework.org/manual/#creating-dataflow. Accessed: 2021-04-11.

[3] Error prone bug patterns: Immutable. https://errorprone.info/bugpattern/Immutable. Accessed: 2021-02-28.

[4] HashMap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. Accessed: 2021-03-19.

[5] Java Language Specs: Enum classes. https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.9. Accessed: 2021-02-24.

[6] Java Virtual Machine Specification: Arrays. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-3.html#jvms-3.9. Accessed: 2021-02-27.

[7] The Checker Framework Manual: Custom pluggable types for Java. https://checkerframework.org/manual/. Accessed: 2021-03-19.

[8] Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual Nullable Types for Java. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 87–97. ACM, 2017.

[9] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[10] Dave Clarke, James Noble, and Tobias Wrigstad, editors. *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*. Springer-Verlag, Berlin, Heidelberg, 2013.

[11] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. Test cases of Glacier. https://github.com/mcoblenz/Glacier/tree/master/tests/glacier. Accessed: 2021-02-20.

[12] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. Glacier: transitive class immutability for Java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE, 2017.

[13] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring Language Support for Immutability. ICSE '16, page 736–747. ACM, 2016.

[14] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 681–690, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable Static Inference for Generic Universe Types. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 333–357, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] Michael D. Ernst and Mahmood Ali. Building and using pluggable type systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 375–376, 2010.

[17] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 2014.

[18] Reiner Hähnle and Marieke Huisman. 24 challenges in deductive software verification. In *ARCADE@ CADE*, pages 37–41, 2017.

[19] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. A Unified Lattice Model and Framework for Purity Analyses. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 340–350, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and Checking of Object Ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, page 181–206, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. OOPSLA '12, page 879–896, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Laurent Hubert. A Non-Null Annotation Inferencer for Java Bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, page 36–42, New York, NY, USA, 2008. Association for Computing Machinery.

[23] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative Pluggable Types for Java. *ACM Trans. Program. Lang. Syst.*, 32(2), February 2010.

[24] Bertrand Meyer. *Object-oriented software construction*. Interactive Software Engineering (ISE) Inc., 2010.

[25] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. volume 1445, 07 1998.

[26] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 201–212, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Alexander J. Summers and Peter Müller. Freedom before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 1013–1032, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Mier Ta. Context Sensitive Typechecking And Inference: Ownership And Immutability. UWSpace, 2018.

[29] Matthew S. Tschantz and Michael 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*, OOPSLA '05, page 211–230, New York, NY, USA, 2005. Association for Computing Machinery.

[30] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise Inference of Expressive Units of Measurement Types. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[31] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 598–617. ACM, 2010.

APPENDICES

Appendix A

Result Table of Possible Use of Fields

This table shows all possible outcome of using different qualifier on field with different instantiation bound and enclosing class.

Legend Columns of the table:

- C.B. = Class bound annotation
- F.D.A. = Field declaration annotation
- F.C.B. = Field's class bound annotation
- F.C.C. = Field's constructor call during initialization

The order of columns follows the order they "appear" in code:

```
@CB
class Clazz {
    @FDA ClassWithBoundFCB field = new @FCC ClassWithBoundFCB();
    static void foo(@ClassUse Clazz c) {
        c.field; // field access result
    }
}
```

Note that the class declarations of "invalid" rows are invalid, so no use of the class will make sense, thus the corresponding rows are "N/A".

C.B.	F.D.A.	F.C.B.	F.C.C.	Class Use	Field Access Result
M	RDM	M		M	M
				RO	RO (transitive)
		RDM	M	M	M
				RO	RO (transitive)
			I	N/A	Invalid init - Incompatible with field decl
			RDM	N/A	Invalid init - Incompatible with field decl
			(none)		
		I		N/A	Invalid init - Incompatible with field decl
	M	M		M	M
				RO	M
		RDM	M	M	M
				RO	M (non-transitive)
			I	N/A	Invalid init - Incompatible with field decl
			RDM	N/A	Invalid init - Incompatible with field decl
			(none)		
		I		N/A	Invalid init - Incompatible with field decl
	I	M		N/A	Invalid init - Incompatible with field decl
		RDM	M	N/A	Invalid init - Incompatible with field decl
			I	M	I
				RO	I
			RDM	N/A	Invalid init - Incompatible with field decl
			(none)		
		I		M	I
				RO	I
I	RDM	M		N/A	Invalid init - Incompatible with field decl
		RDM	M	N/A	Invalid init - Incompatible with field decl
			I	I	I
				RO	RO
			RDM	I	I
				RO	RO
			(none)	(any)	Bottom (null)
		I		N/A	Invalid init - Incompatible with field decl
	M	M		I	M
				RO	M (non-transitive)
		RDM	M	I	M
				RO	M (non-transitive)
			I	N/A	Invalid init - Incompatible with field decl

			RDM	N/A	Invalid init - Incompatible with field decl
			(none)	N/A	Invalid class decl - all fields must initialized
		I		N/A	Invalid init - Incompatible with field decl
	I	M		N/A	Invalid init - Incompatible with field decl
		RDM	M	N/A	Invalid init - Incompatible with field decl
			I	I	I
				RO	RO
			RDM	I	I
				RO	I
			(none)	N/A	Invalid class decl - all fields must initialized
		I		I	I
				RO	I
RDM	RDM	M		N/A	Invalid class decl - invalid anno
		RDM	M	N/A	Invalid init - Incompatible with field decl
			I	N/A	Invalid init - Incompatible with field decl
			RDM	M	M
				I	I
				RDM	RDM
				RO	RO
			(none)		
		I		N/A	Invalid init - Incompatible with field decl
	M	M		M	M
				I	M
				RDM	M (non-transitive)
				RO	M (non-transitive)
		RDM	M	M	M
				I	M
				RDM	M (non-transitive)
				RO	M (non-transitive)
			I	N/A	Invalid init - Incompatible with field decl
			RDM	N/A	Invalid init - Incompatible with field decl
			(none)	N/A	Invalid class decl - all fields must initialized
		I		N/A	Invalid init - Incompatible with field decl
	I	M		N/A	Invalid init - Incompatible with field decl
		RDM	M	N/A	Invalid init - Incompatible with field decl
			I	M	I
				I	I

				RDM	I
				RO	I
			RDM	N/A	bottom (null)
			(none)		
		I		M	I
				I	I
				RDM	I
				RO	I

Table A.1: Possible uses of fields under all condition

Appendix B

Repository of Experimented Projects

The repository URLs of the projects we experimented with PICO is listed below.

Project Name	Repository URL
imgscalr	https://github.com/rkalla/imgscalr.git
jama	https://github.com/topnessman/jama.git
react	https://github.com/topnessman/react.git
exp4j	https://github.com/fasseg/exp4j.git
ECC-RSA-Backdoor	https://github.com/topnessman/ECC-RSA-Backdoor.git

Table B.1: URLs of the repository exprimented with PICO