Context-Sensitive Optional Type Systems Meet Generics: A Uniform Treatment and Formalization

by

Haifeng Shi

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, 2023

Author's Declaration

Abstract

This thesis explores a novel design of context-sensitive optional type systems which supports generics. Optional type systems, as the name suggests, optionally enforce type rules in typechecking and can be switched on only when needed. They give flexibility to developers because, whenever they need optional type properties, they can plug these properties into the compiler and perform additional rounds of semantic analyses. Context-sensitive type systems help to resolve a declared type to different types according to usage contexts.

Type parameters and type variable uses can be annotated with optional type qualifiers, and these type qualifiers can impact the process of type argument substitution and context-sensitive resolution of types. This thesis shows (1) existing optional type systems either do not support generics or have type safety issues when context sensitivity is involved; (2) type variables and monomorphic types are treated differently in the type check and type inference modes by these type systems. This thesis formalizes a Java-like language, introduces a new qualifier `Sub` to clarify the semantics of unannotated type variables, and presents a new viewpoint adaptation rule to model how context-sensitive type systems interact with generics. As a concrete instantiation, a case study is conducted in a context-sensitive type system called Practical Immutability for Classes and Objects (PICO).

PICO offers reference and object immutability to Java, making Java programs more robust in areas like thread-safe programming and aliasing control. This thesis also gives an overview of PICO, identifies some interesting issues, and fixes them. Last, this thesis formalizes the core parts of non-generic version of PICO, providing the well-formedness rules, type rules, and operational semantics.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Werner Dietl for his invaluable mentoring throughout my master study. He has been patient and always encouraged me to explore problems that interested me. Only with his expertise, guidance and support, this thesis is possible.

I would like to extend my appreciation to my thesis readers, Professor Arie Gurfinkel and Professor Mahesh Tripunitara for their insightful feedback.

I am thankful to all my colleagues, Alex, Zhiping, Alice, Aosen, Piyush, Florian and Di for the inspiring discussions, help and support. I also want to say thank you to John, Jane and all my friends met in UWaterloo who brought me happiness and made me a better person.

My deepest gratitude to my parents and my brother. Without their support, this wonderful research journey would be impossible.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 EISOP Checker Framework 1

1.2 Context Sensitivity 2

1.3 Limitation in Checker Framework Generics 3

1.4 Practical Immutability for Classes and Objects 4

2 Background and Related Work 6

2.1 Generics component of Checker Framework 6

2.1.1 Support of Parametric Polymorphism 6

2.1.2 Important Features in Checker Framework Generics 7

2.2 Immutability 8

2.2.1 Definition 8

2.2.2 Immutability in Mainstream Programming Languages 9
2.2.3 Literature Review 10
2.2.4 Observations 16
2.2.5 Other Applications 17

3 Practical Immutability for Classes and Objects 18
3.1 Language Features 18
3.1.1 Type Qualifiers 19
3.1.2 Abstract State 23
3.1.3 Transitive Immutability 26
3.1.4 Receiver-Context-Sensitivity 27
3.1.5 Assignment-Context-Sensitivity 29
3.1.6 Defaulting Mechanism 30
3.2 Improvements 32
3.2.1 Restricting Type-use Locations of Qualifiers 33
3.2.2 Warn Redundant Annotations 34
3.2.3 Apply Flow-Refinement for Receivers when Computing the Type of LHS 34
3.2.4 Adding the Lost Type Qualifier to the System 36
3.2.5 Improve Typecast Logic 37
3.2.6 Improvements to Transitive Immutability 39
3.2.7 Other Improvements 40
3.3 Formalization 42
3.3.1 Helper Functions 42
3.3.2 Syntax 43
3.3.3 Subtype & Subclassing rules 44
3.3.4 Well-Formedness Rules 44
3.3.5 Static Type Checking 46
3.3.6 Operational Semantics 48

4 Generics with Context-Sensitive Optional Type Systems **51**

4.1 Exploring PICO-Generics 51

4.2 The General Problem and Solution 53

4.2.1 The General Problem 53

4.2.2 The Solution 54

4.3 Formalization 55

4.3.1 Syntax 56

4.3.2 Viewpoint Adaptation and Type Variable Substitution 57

4.3.3 Subtyping & Subclassing Rules 59

4.3.4 Well-Formedness Rules 61

4.3.5 Static Type Checking 62

4.3.6 Operational Semantics 64

4.4 Case Study in Immutability 66

4.4.1 Annotated Type Variable Uses 66

4.4.2 Inference of Unannotated Type Variable Uses 68

4.4.3 Simple Constraint Generation Rules 69

5 Conclusion and Future Work **71**

References **73**

List of Figures

2.1 Viewpoint adaptation rule for field access 12

3.1 Lattice for mutability types 19
3.2 Memory diagram for listing 3.3 36
3.3 Typecast errors/warnings 38

List of Tables

3.1 Combinations of Assignability and Mutability 24
3.2 Differences between explicit and implicit annotations 25

Chapter 1

Introduction

The famous phrase "well-typed program cannot go wrong" [21] underscores the importance of static type systems to running programs. Compared with weakly-typed languages, strongly-typed languages are able to discover more errors at compile-time, raising problems much earlier, and thus reduce the tremendous human effort spent in fixing runtime exceptions.

Different programming languages have different type systems. For example, Kotlin offers a nullness type system which can effectively eliminate Null Pointer Exceptions (NPE); Rust provides an ownership type system which benefits the memory management. What if developers want to migrate some type properties from one language to another language or enhance the existing one with more type properties? Directly developing on built-in type system of these popular languages is hard and time-consuming. A solution is to treat these advanced properties as plug-ins, rather than the mandatory part of the built-in type system, and whenever these properties are needed, we attach them to the built-in type system. These properties form the pluggable type system [4], and for different purposes we can select suitable pluggable type systems to participate in the type checking phase, which offer flexibility and advanced type safety to our program.

1.1 EISOP Checker Framework

The EISOP Checker Framework [10] is a well-maintained and popular framework for assisting developers building pluggable type systems. Some included type checkers [7] built on top of the Checker Framework are Nullness Checker (to detect NPEs in programs), Interning Checker (checks for the proper use of reference equality), Tainting Checker (prevents the leak or corruption of data), Lock Checker (detection of concurrency errors), PUnits [34] (for expressive units of measurement types and inference) and cryptography API usage checkers [19][35] (to ensure correct usages of them).

In the implementation of the Checker Framework, these pluggable type checkers are essentially Java annotation processors, which define the optional types as Java annotations, annotate/infer annotations in the source code and process these annotations on Abstract Syntax Tree (AST) generated from the syntax analysis (the process of annotations corresponds to the type-check phase). Let us look at a simple example in the Lock Checker.

```
1 @GuardedBy("lock") Object x = new Object();
2 static void test() {
3     x.toString(); // type check fails.
4     lock.lock();
5     x.toString();  // okay.
6 }
```

Lock Checker defines the `@GuardedBy({})` annotation and implements related type rules to enforce that certain lock(s) must be held when operation occurs. On line 1, variable `x` is declared with two types, `Object` and `@GuardedBy("lock")`. This pluggable type checker demands the dereference of `x` must require `lock` to be held, thus line 3 fails and line 5 type checks.

Another interesting application of the EISOP Checker Framework is its joint use with deductive verification tools (e.g. KeY [1], JML [18]), which takes advantage of the scalability of the type systems and precision of the verification analysis [17].

1.2 Context Sensitivity

Context sensitivity is a language feature provided in the EISOP Checker Framework, and it assists discovering accurate types without being too conservative. It helps to resolve a declared type to different types according to distinct usage contexts. Context sensitivity can be encoded by viewpoint adaptation laws, which adapts a type from a viewpoint of another type (the type of the usage context). Below shows a simple usage of viewpoint adaptation in Checker Framework .

```
class C {
    @RDQ Object f;
    static void test(@X C x, @Y C y) {
        @X Object o1 = x.f;
        @Y Object o2 = y.f;
    }
}
```

RDQ stands for receiver-dependent-qualifier which suggests the type of the field depends on the type of the receiver. On line 4, we adapt the type of f from the viewpoint of x and get type @X back. Similarly, we get type @Y back on line 5. As a result, we can see field f is resolved to different types at two contexts. A simplified version of viewpoint adaptation rule is provided below:

$$q \rhd \text{RDQ} = q \tag{1.2.1}$$

$$\_ \rhd \ q = q \text{ (otherwise)} \tag{1.2.2}$$

For the first rule, whenever we adapt a field typed with RDQ from the viewpoint of type q we get q back; for the second rule, if the field is not RDQ, any type looks at it gives back q. Field access is the typical usage context using viewpoint adaptation, but not the only one. Viewpoint adaptation can also occur in other contexts, such as method invocation and object creation.

Apart from Checker Framework, context-sensitivity is adopted in many other languages as well as static analyzers. SFlow [14] is a context-sensitive type system for secure information flow and SFlowinfer is its inference counterpart. DFlow [20] is an another context-sensitive integrity type system which can detect information flow vulnerabilities in web and Android applications.

1.3 Limitation in Checker Framework Generics

Existing generics component in Checker Framework cannot work well when context sensitivity (viewpoint adaptation) comes into play. Take the code below for example, what would be the type of d.element on line 8? (1) If we only think about substitution, then d.element will be resolved to type @B Element and would raise an error because @A, @B are incomparable. This is because the whole declared field type @A E is substituted by the type argument @B Element. This is not consistent with how Checker Framework deals with other type systems without viewpoint adaptations. For example, the nullness type system make the annotated type variable field shadow the annotations provided in the type arguments (background section 2.1.2 discusses this in details [1]). (2) If we only think

[1]The behavior of substituting the whole declared field type along with the annotations with the type arguments, is colored by Generic Universe Types [8], which itself does not allow annotations on type variable fields at first place. However, Checker Framework does allow such option which results in an unclear semantics when doing viewpoint adaptation.

about viewpoint adaptation, we will use the receiver type to look at the declared field type and produce the adapted type, which would be `@A`. Hence, no error reported.

```
class Demo<E> {
    @A E element;

    static void test() {
        Demo<@B Element> d = new @A Demo<@B Element>();
        // d.element = @A |> @ A = @A
        // thus expects no error here
        d.element = new @A Element();
    }
}
class Element {}
```

In this discussion, type argument substitution and viewpoint adaptation produces the opposite types and contradicts with each other, so how should we handle them nicely? More, if a generic field is not annotated, what would be the optional type of that field? For annotated type parameters and type variable uses, what is the precise semantics for them? These are questions worth considering. This thesis tries to clarify the semantics and model how context-sensitive type systems interact with generics.

1.4 Practical Immutability for Classes and Objects

Immutability is one of the most discussed properties when designing the type system. Immutability, as the name suggests, can guarantee a program component (e.g., field, local variable) to be unchanged in the execution of the software. This property is crucially desired when building a correct and secure program. For example, immutable variables are not prone to race conditions in multi-threading applications. More, programs which have immutability guarantees are considered more secure. An example is shown in the listing below.

```
protected ElectionResults count(Votes result) { ... }
```

In the example, we want to make sure the integrity of the votes, preventing it from being corrupted in the `count` method. If we can make variable `result` to be immutable, then we ensure the implementation of the method will not violate the contract.

Though functional programming languages like Haskell or multi-paradigm languages like Ocaml by default enforce immutability, more widely used imperative ones do not. For example, `Java` cannot declare a field or object immutable. The `final` keyword only helps with assignability and `private` modifier does not solve the aliasing issue (another alias outside the class could still mutate the object the private field points to). Because of that, some researchers investigate means to plug immutability properties into Java's type system. ReIm & ReImInfer [15], for example, assists programmers to annotate Java programs with immutability annotations and typechecks if there are immutability violations. Some other tools have been invented for this purpose, but their complexity and flexibility limits them from being applied to real-world large software projects. A literature review at the background section 2.2 tries to illustrate the trade-offs and distinctions of these tools.

Practical Immutability for Classes and Objects (PICO for short) [30] [31] is a context-sensitive immutability type system built on top of Checker Framework. PICO is practical and flexible by having Context Sensitivity, Abstract State, Transitive Immutability, Reference/Object Immutability and limited immutability polymorphism. This thesis identifies interesting issues in PICO and fixes them. Furthermore, a formalization of PICO, including type rules and operation semantics is presented in the thesis.

Thesis Contributions

- A novel viewpoint adaptation/type variable substitution logic with clear semantics for context-sensitive optional type system interacting with generics. To support the theory, we provide a formalization of it and carry out a case study in immutability.
- An simplified inference approach benefited from the uniform treatment of type variables.
- A literature review on immutability, which illustrates important definitions in the domain, compares immutability in different languages and discusses several related articles.
- A review of the immutability type system, PICO. Then I show some improvements as well as the formalization to the type system.

Thesis Outline Chapter 2 provides essential background knowledge to assist readers to understand this thesis. Before discussing our model of viewpoint adaptation interacting with generics in chapter 4, we first discuss PICO in chapter 3 to familiarize readers with the immutability system. Note that these two chapters are still independent of each other. Chapter 5 concludes this thesis and discusses about future directions.

Chapter 2

Background and Related Work

In this chapter, we first the present background knowledge of EISOP Checker Framework [10], the framework for building customized optional type system, focusing on the generics component of it. Then we give a literature review in immutability.

2.1 Generics component of Checker Framework

2.1.1 Support of Parametric Polymorphism

The support of parametric polymorphism can slow down the code analysis process but helps greatly in code maintenance and reducing code duplication. Checker Framework fully supports Java generics, which sets it apart from other static analysis tools, such as NullAway [2]. NullAway is a fast, lightweight, and popular type-based static analysis tool aiming at eliminating `NullPointerException`(NPE) in Java programs. Programmers distinguish references which can hold null values and those cannot, by annotating them to be `@NonNull` or `@Nullable`. However, NullAway ignores annotations on type parameters and type arguments, and only allows to annotate fields in order to make them nullable. This could introduce code duplication. Below is an example annotated by NullAway. As we can see, we need three additional generic classes to express the semantics having a nullable key or nullable value, which defect the purpose of using the generic Pair class.

```java
class PairNullableKey<K, V> {
    @Nullable K key;
    V value; // default to be @NonNull to preserve soundness
}
```

```java

class PairNullableValue<K, V> {
     K key;
    @Nullable V value;
}

class PairNullableKeyAndValue<K, V> {
    @Nullable K key;
    @Nullable V value;
}
```

In Checker Framework, it is easy to express that by declaring a `Pair<@Nullable String, Integer> p` or `Pair<String, @Nullable Integer> p2`. Moreover, the way of annotating generic fields in NullAway makes it hard for code relying heavily on the third-party libraries, as discussed at section 6 in the NullAway paper [2].

2.1.2 Important Features in Checker Framework Generics

There are some important features worth mentioning in Checker Framework generics.

2.1.2.1 Restricting the instantiation of a generic class

By annotating the type parameter and the extends clause we can enforce additional constraints to the instantiation of the generic class. For example,

```java
class Box<@NonNull T extends @Nullable Object> { ... }
```

`@NonNull`, `@Nullable` are two types from the nullness type system, and `@NonNull` is the subtype of `@Nullable`. By adding the annotations in this way, `T` is not only the subtype of `Object`, but also the subtype of `@Nullable`. In this way, we enforce additional constraints to the type arguments. For wildcard restriction, it is similar to the above approach. For example, we can declare a local variable like this: `Box<@Nullable ?> b`.

2.1.2.2 Annotating on a use of a generic type variable

In a use of a generic type variable, for instance, when declaring a generic field, we can also add our optional type information on it.

```java
class Box<T> {
    ...
    @Nullable T f;
    ...
    static void test() {
        Box<@NonNull Object> b1 = new Box<>();
    }
}
```

A question would arise in the substitution phase, what would be the type of `b1.f`? As in the type variable substitution phase, `T` will be replaced by `@NonNull Object` supplied by the local variable `b1`. However, `T` is annotated with `@Nullable`. Should we prioritize the declaration-site annotation or the use-site annotation? The answer here is that annotation on the declaration will shadow the annotation on the type argument (but Checker Framework handles this inconsistently when viewpoint adaptation is involved, and we have discussed it in section 1.3).

2.2 Immutability

The review proceeds as follows. Section 2.2.1 introduces important concepts. Section 2.2.2 discusses immutability in mainstream programming languages. Section 2.2.3 is the paper review part, discussing 5 tools in this area of research. Section 2.2.4 states our observation after reviewing these tools. Section 2.2.5 shares some applications in the immutability domain.

2.2.1 Definition

The definition of immutability may not be as straightforward as it seems. Here are some important definitions in the field of immutability.

- Readonly Reference: Readonly reference cannot be used to mutate the object to which it refers. However, if the object has other mutable references, the object can still be mutated.
- Immutable Object: An immutable object cannot be mutated once it finishes the construction. Unlike readonly reference, immutable object cannot be mutated through any references.

- Immutable Class: An immutable class means that all instances of the class are immutable.
- Transitive Immutability: All fields can reach through an immutable object cannot be mutated and reassigned in transitive immutability.
- Non-Transitive Immutability: Fields directly reachable through an immutable object cannot be mutated and reassigned, but fields transitively reachable through an immutable object can still be mutated. Listing 2.1 shows the difference between transitive immutability and non-transitive immutability.
- There are also some other dimensions of immutability, interested readers can refer to Exploring Language Support for Immutability [6] for more details.

```
class A {B b; int i;}
class B {
    int j;
    static void modify(/* Immutable */ A a) {
        a.i++; // disallowed in non-transitive and transitive
   immutability
        a.b.j++ // allowed only in non-transitive immutability
    }
}
```

Listing 2.1: Illustration of transitive and non-transitive immutability

Sometimes, people confuse assignability with immutability. For example, in Java, a reference declared with `final` means that the reference cannot be reassigned to point to another object. However, this does not mean the object it points to cannot be mutated. On the other hand, if an object is believed to be immutable, that does not mean the reference pointing to that object cannot be reassigned to point to another object. Last, though assignability and immutability are independent of each other, an instance of `class A` is at least non-transitive immutable if all fields of `A` are `final`.

2.2.2 Immutability in Mainstream Programming Languages

Java Java provides the `final` keyword to declare a reference that cannot be reassigned. It can not prevent the object to which it refers from being mutated.

C C/C++ languages provide the `const` keyword to specify immutability. A `const` variable is not modifiable. However, there are some pitfalls in using the keyword. First, a `const` method only prevent it from modifying the receiver but not other variables. So a `const` method is not a pure method. Second, an unchecked cast can remove the `const` from a variable.

JS Apart from having the `const` keyword, JavaScript provides freeze method which can make a mutable object immutable.

Golang `const` keyword specifies compile-time constant, and can only be numbers, characters, strings or booleans. That means, a constant cannot be initialized in the `init` function.

Rust Rust takes step further compared with Java, C++. In Rust, a borrow or binding is by default immutable, unless specifying it with `mut` keyword. This helps to achieve the primary goal of Rust: Safety.

Haskell Languages like Haskell, Ocaml default all variables to be immutable.

During surveying different languages' immutability support, we found that new languages such as Rust, Swift, Kotlin, Scala treat immutability at least on par with mutability, compared with old languages like C, Perl providing little immutability features. This shows modern software development has more safety demands, and we can trade easier abstraction for higher immutability guarantees [25].

2.2.3 Literature Review

The papers are ordered by the year it published. For each paper, we first briefly go over its main ideas, contributions and then focus on its design and evaluation. Our discussion and comparisons are placed at the last part 2.2.4.

2.2.3.1 Javari

Summary of Javari Javari [33] separates the notion of mutability and assignability, and introduces the concept of abstract state. Users (of javari) can put annotations (mutable,assignable) to classes, fields, and methods' receivers to prevent an immutable item from being updated or an unassignable item from being assigned. Its key guarantees are reference immutability and transitive immutability. The formalization of Javari is similar to the formalization in Featherweight Generic Java (FGJ) [16]. Other contributions in this paper: (a) the language supports generics and multi-dimensional array. (b) supports mutability polymorphism. (c) supports reflection and serialization. A follow-up work introduces Javarifier [23] which does the inference of reference immutability in Javari.

Design of Javari

1. Immutability Type Hierarchy: `readonly` is the top type in the type system.
2. Assignability: In this paper, `final` declares a field to be non re-assignable, while `assignable` specifies a field can always be reassigned even if the receiver is a read-only reference.
3. Field Mutability: The adaption of the type of field is the key part of this paper, called as this-assignable and this-mutable, which means the field type is determined by the reference referring to it. For example, if the reference to it is read-only, then the field type is read-only and unassignable, or the field is mutable and assignable if the reference accesses it is mutable. Figure 3 in the Javari paper shows how to resolve `a.b` under different Javari keywords.
4. Abstract State: In order to capture full abstract state, Javari enforces deep-immutability. Mutation is any modification of the object's abstract state. The type system also provides an option to exclude certain fields from the object's abstract state.
5. Generics and Arrays: The use of type parameter within the generic class will have the same mutability type of the instantiation of the generic type. For generics, Javari limits the annotation of mutable on type parameter, as this would cause statically non-verifiable downcasts. For arrays, user can specify mutability in each level of the array, but arrays are Javari is not co-variant.

Evaluation of Javari Javari is tested using more than 160,000 lines of code.

Discussion (1) Adopting the uses of this-assignability, this-mutability reduces the annotation burden of programmers; (2) Javari has backward compatibility with Java, which offers convenience to Java users; (3) whether reference immutability is sufficient for software projects or not is a design choice worth considering.

2.2.3.2 ReIm

Summary of ReIm & ReImInfer ReIm [15] is related to Javari, as their type hierarchies are quite similar, and they both enforce transitive immutability and reference immutability. ReIm has advantages and disadvantages compared with Javari. Overall, it is simpler but less expressive. As claimed by the authors, the primary application of ReIm is for method purity analysis, thus it can drop some language features with no harm. ReImInfer is the inference tool to ease the use of ReIm, by using set-based approach to derive possible solutions for method formal parameters, declared method receivers, method returns, etc. The inference system is claimed to scale linear in practice and the worst time complexity is $O(n^2)$ (n is the size of the program, the number of parameters, local variables...).

Design of ReIm

1. Context-Sensitivity: ReIm provides two kinds of context-sensitivities, one is (a) receiver-sensitivity, which lets the type of a field adapt to the type of the receiver. See figure 2.1 for the viewpoint adaptation rule. Another one is (b) calling-sensitivity, which determines the type of right-hand-side of an assignment by acquiring the type of left-hand-side. Both kinds of sensitivities require the annotation `polyread`.

$$\begin{array}{rcl} \_ \rhd \mathsf{mutable} & = & \mathsf{mutable} \\ \_ \rhd \mathsf{readonly} & = & \mathsf{readonly} \\ q \rhd \mathsf{polyread} & = & q \end{array}$$

Figure 2.1: Viewpoint adaptation rule for field access

Design of ReImInfer

1. Set-based approach for inference. The main idea is to use a map from variables/references to possible types. During iterations, if the assignment of the type qualifiers cannot typecheck, the corresponding type qualifiers will be removed from the set. The algorithm is deterministic and guaranteed to find solutions or end with errors.

2. The paper uses "preference ranking" to maximize the number of readonly references in the program.

3. Can be applied to method purity analysis. When a method is believed to have a mutable `this` reference or mutable arguments, the method is regarded as impure. When taking static field into consideration, the paper invents three immutability qualifiers for methods, namely, `readonly`, `polyread` and `mutable`. As long as the method is not mutable (that is, polyread or readonly), the method is considered to be pure.

Evaluation We list three main parts of the experiment here.

1. Efficiency: as shown in figure 6 in the original paper, when the repository size is under 100k LOC, two tools have similar timing results. However, ReImInfer clearly scales better compared with Javarifier when the project size increases.
2. Correctness: the differences between the two tools are minor, most of which can be contributed to different semantics of the two tools.
3. Purity inference: overall, ReIm and ReImInfer perform better than JPPA [28], JPure [22] on different benchmarks.

Discussion We think this tool is less expressive but more lightweight compared with Javari. More, ReImInfer is built on top of Checker Framework, which inherits the advantages of pluggable type systems.

2.2.3.3 Glacier

Summary of Glacier Glacier [5], short for "Great Languages Allow Class Immutability Enforced Readily", is built on top of Checker Framework, and is a pluggable immutability type system for Java. Glacier decides to support transitive immutability, class immutability and reference immutability after conducting interviews with senior developers. In general, this tool is even simpler than ReIm, and its main purpose is for practicality and ease of use. In evaluation, a user case study is also provided to show the superiority of Glacier.

Design of Glacier

1. Type Hierarchy: It only contains two class qualifiers, `MaybeMutable` and `Immutable`.

2. Support Class Immutability: User can choose to annotate a class or not. If a class is annotated, all instances of the class should be in the same qualifier as the class (there is no subtyping relationship). For an immutable class, its subclasses should all be immutable, and all fields of the class should be immutable. For an MaybeMutable class, it can have immutable subclass. For a use of type whose declaration has no annotation, it defaults to be `MaybeMutable` type.

3. Array: This paper adopts two more qualifiers, one is `Readonly`, another is `GlacierBottom` to support annotating arrays. The first one is used on method parameters, so the method can accept both mutable and immutable arrays and guarantee not changing them in the method. Though not explicitly said in the paper, I assume readonly is the top qualifier in the type system. The second one is used on clone method return, so that the bottom qualifier can be assigned to any other qualifiers in the type system.

4. Simplicity: Glacier has no run-time component, no polymorphic qualifier, no inference counterpart, which makes Glacier as simple as possible.

Evaluation

1. Case study on ZK Spreadsheet [36]: Showing that Glacier can be applied to read-world complex software.

2. Case study on Guava ImmutableList [13]: The only aspect Glacier cannot cover is ImmutableCollection, because of the lack of support for lazy initialization.

3. User study: 20 experienced programmers, with a mean of 9.5 years of experience of programming were chosen in this survey. They were briefly taught with how to use Glacier and how to use `final` keyword (as suggested in Effective Java) before doing the tasks. In the end, the authors claim that Glacier is easier to use and have better immutability enforcement. Table I and II in the paper shows the experiment results.

Discussion

1. Simple design but also applicable: this tool is the easiest to use among all other tools discussed in this review, and successful applications on two open-source projects demonstrate the practicability of this tool.

2. No support for lazy initialization. Some object initialization may be triggered by the first access from the client. No support for this limits the usability of the tool.

2.2.3.4 CIFI

Summary of CIFI CIFI [26] stands for "Class and Field Immutability Analysis". The novelty comes from the additions of class and field immutability which are important and can simplify the design of the tool. Based on OPAL, a static analysis framework, CIFI can infer the properties of class and field immutability defined in the model.

Design of CIFI

1. Separation of assignability and immutability: In CIFI, there are two lattices, field assignability lattice and immutability lattice. Despite having two lattices, immutability relies on assignability. In Definition 6, it says "A field is mutable if and only if it is assignable". In implementation field immutability needs the analysis result of field assignability, as shown in figure 3 of the paper.

2. Fine-grained lattice for assignability: The papers offers more fine-grained lattices by enumerating all conditions. For example, a final field is called as non-assignable in CIFI; a non-final field but having no method to re-assign the field is called effectively non-assign; Lazily initialized refers to a non-final field initialized after construction.

3. Fine-grained lattice for immutability: For immutability, there are mutable, non-transitively immutable, dependently immutable and transitively immutable. Dependently immutable is to model the effect of generic types on immutability, and transitively or non-transitively immutable is based on the runtime type of generic type T. To determine whether the field is which one of the two, the analyzer goes through all places of instantiation of the generic type.

4. Class immutability: As said earlier, the paper focuses on class immutability rather than immutability of instances. For the inference rule, the immutability of a class is the least upper bound of all its instance fields.

Evaluation CIFI either correctly produces or soundly over-approximates the annotations on over 470 test cases. As the model of CIFI considers non-transitive and transitive immutable, it is claimed to be better than Glacier.

Discussion CIFI is based on OPAL, which benefits from some existing analyses. For example, it uses escape analysis to determine if a field is "effectively non-assignable" or not.

2.2.3.5 Error Prone Immutable

Summary Google Error Prone [11] statically analyzes and catches common programming errors before run-time. It is wildly used, and is incorporated in Google's Java build system. It has a type declaration annotation `@Immutable`, to be used on type declarations. Error Prone validates that `@Immutable` classes and interfaces are deeply immutable by checking these constraints:

1. All fields should be final.
2. Fields should also be in immutable type, or null.
3. Guarantee that `this` pointer does not escape construction.
4. An immutable subclass cannot have a mutable superclass.

This directly leads to several limitations of the tool:

1. Does not support lazy-initialization and circular initialization.
2. Only class-immutability is supported. User cannot specify the mutability of an instance.
3. Array type is considered mutable in Google Error Prone. No array field allowed in the immutable class.

2.2.4 Observations

- All tools discussed in this review achieve deep immutability, also known as transitive immutability, while CIFI supports both transitive immutability and non-transitive immutability.
- Immutability tools becomes simpler and more practical. From Javari to ReIm and to Glacier, a frequently asked question is "Is this tool easy to use?". Complex tools can be powerful but developers with no static analysis knowledge may fail to appreciate the capability of these tools. Simplicity is gained by giving up some features. For example, Glacier gives up on lazy initialization, abstract immutability and immutability polymorphism; ReIm gives up on abstract immutability, generics.
- Static analysis frameworks are popular when designing these tools. ReIm and Glacier use Checker Framework, and CIFI uses OPAL.

2.2.5 Other Applications

A type system [12] that models a prototype extension to C# presents how to combine uniqueness and reference immutability for safe parallelism. The type system is flexible and supports polymorphism over type qualifiers. Uniqueness and immutability are jointly used in another article [27], implemented in language L42, and supports information flow control with type modifiers. Immutability can also assist guaranteeing the aliasing safety [24].

In industry, threats such as ransomware have now become so prevalent that many object-storage products support data immutability. As the name implies, immutable data cannot be deleted or modified. An update usually would produce a new version of that file while retaining the old one. For example, Amazon S3 can be set up and configured to work with immutable storage buckets.

Chapter 3

Practical Immutability for Classes and Objects

Practical Immutability for Classes and Objects (PICO) enhances Java's type system by offering reference/object immutability to it. The tool fully supports Java generics and immutability polymorphism; The implementation is plugged into the EISOP Javac [32] which uses Java's annotation processor to traverse program's abstract syntax tree and control flow graph for flow-sensitive program analysis. Some related applications lie in thread-safe programming, prevention of information leak, aliasing control and elimination of defensive copying.

3.1 Language Features

Before going to the formalization part, let's take a closer look at PICO's features. PICO's design focuses on balancing flexibility, safeness and usability. It has carefully designed defaulting mechanism and supports transitive immutability, abstract state, context sensitivity, reference/object/class immutability and leverages initialization checker to handle initialization problems. Even though there are two theses [30] [31] on this topic, three reasons make contributing and improving this type system and its implementation hard.

- Two theses are well written but occasionally diverge from each other. Theory in the theses sometimes is inconsistent with the implementation. Visitor pattern used in the framework makes adding and implementing functionalities easy but debugging hard. This introduces difficulties when analyzing the inconsistency.

- The design space for PICO (or immutability) is large and some design choices are subtle and take a lot of time discussing.
- PICO implementation is built on top of Checker Framework which has 400k lines of code and is frequently evolving (Around 1000 commits were merged in the main branch in year 2020). We try to integrate PICO into the Checker Framework's downstream CI pipeline but the implementation initially used 2018-version Checker Framework for developing and that makes refactoring hard. Also, there are also some bugs in Checker Framework need to be addressed in order to make PICO working.

In order to address the first two difficulties, I try to provide a thorough and up-to-date description of PICO before introducing the formalization.

3.1.1 Type Qualifiers

The lattice for this mutability type system is as follows:

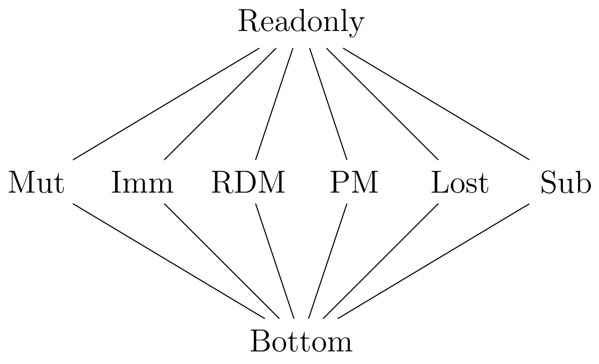


Figure 3.1: Lattice for mutability types

3.1.1.1 Readonly

- Semantics: The top qualifier of the lattice, the least upper bound of mutable and immutable types. Readonly qualifier only applies on references which can point to a mutable or immutable object. Developer can never use a Readonly reference to mutate an object, but that does not guarantee the object cannot be mutated by other aliases.

- Default: All local variables are default to be Readonly and subject to flow refinement. Implicit upper bound of a type parameter is Readonly.
- Use Locations: Cannot be annotated on classes, class extends/implements clauses and in object creations.

3.1.1.2 Mutable

Represented by Mut in the lattice.

- Semantics: When used in object creation, it means the object created will always be mutable; when annotating on a reference, it means the object that reference points to is mutable; when applying on a class, it means all instances of that class are mutable. A mutable object can be referred by a Readonly reference, but never an immutable reference due to incompatible subtype relationship.
- Default: Mutable qualifier is the default for most locations and types. We adopt the mutable qualifier as the default to achieve better compatibility with unannotated java programs (where most objects are mutable). A class is by default a mutable class; a type use (except it is a field declaration) is by default mutable if the class of this type is Mutable or RDM.
- Use Locations: No restriction.

3.1.1.3 Immutable

Represented by Imm in the lattice

- Semantics: When used in object creation, it means the object created will always be immutable; when annotating on a reference, it means the object that reference points to is immutable; when applying on a class, it means all instances of that class are immutable. An immutable object can be referred by a Readonly reference, but never an mutable reference due to incompatible subtype relationship.
- Default: All literals, primitive types along with their boxed types are immutable. Enum classes and enum constants are immutable. String, Number, BigDecimal, BigInteger classes are immutable. In fact, these types and constants can only be immutable.
- Use Locations: No restriction.

3.1.1.4 ReceiverDependentMutable

Represented by RDM in the lattice. We will use RDM to represent this type qualifier in this thesis.

- Semantics: As the name suggests, the resolved mutability type depends on the receiver type. RDM is used to achieve context sensitivity. For field access, the field type depends on the receiver type; For method call, the return type, parameter type and type parameter will be adapted from the viewpoint of the caller; For object creation, the type of the constructed object will depend on the type provided in the `new` object creation. For example, `new @Mutable Object();`. When used on a class, it means the class can be instantiated to be immutable and mutable, depending on the type given by the new operation.
- Default: At most of the time, a type use will be default mutable if the type declaration is RDM. Only when the type use is a field declaration, the type use is by default RDM.
- Use Locations: Cannot be applied on local variables. Forbidden to use in static contexts.

3.1.1.5 PolyMutable

Represented by `PM` in the lattice,

- Semantics: PolyMutable adds assignment-sensitivity to the type system. A variable with polymorphism qualifier can be resolved to different types depending on the assignment context (more specifically, depending on the target type of the assignment).
- Default: This qualifier is not the default for any position and type.
- Use Locations: Method parameters, method return type, receiver and object creation

3.1.1.6 Lost

Lost is used to represent a mutability type that cannot be expressed. We will talk more about it in the receiver-context-sensitivity part in section 3.1.4. This new contribution to the immutability type system is inspired by the Generic Universe Types paper [8].

3.1.1.7 Sub

Sub is a new introduced type that serves as the default for generic type uses. With this type, we have a clearer semantics for type variable substitution and viewpoint adaptation. We will talk about this type more at section 4.2.2.

3.1.1.8 Bottom

- Semantics: The bottom type for this type system.
- Default: Default for the implicit lower bound of a type parameter, and for type `null`.
- Use Locations: Only on type parameter lower bounds.

3.1.1.9 Types from the assignability and initialization dimensions

There are some other type qualifiers used implicitly or explicitly in PICO. There are three more types from the assignability dimension: ReceiverDependentAssignable, final and Assignable. We use these types and modifiers to maintain a clean separation of assignability and mutability.

- final: is a keyword provided by Java and can be applied on references. When writing on references, it means that reference cannot be re-assigned after initialization. PICO does not handle final keyword and let Javac enforce related type rules.
- Assignable: the opposite of final, and is used only on fields. Static fields are by default Assignable.
- ReceiverDependentAssignable: RDA for short, which means the assignability of a field depends on whether the receiver is mutable or not. If it is mutable, the field is assignable otherwise final. RDA is only used on and is the default for non-static fields. RDA is only used internally and cannot be written by the programmer.

From the initialization dimension, there are three more type qualifiers. See EISOP Checker Framework manual [10] chapter 3.8.1 for more details. Why does PICO need the Initialization Checker? This is related to the semantics of `Immutable`. An immutable object does not mean it is immutable once it has been allocated on the heap, but it essentially means it becomes immutable once it has completed the initialization. During the initialization, the object will have type `UnderInitialization Immutable` which permits the object to be mutated.

- Initialized: meaning an object is fully initialized.
- UnderInitialization: meaning the object is still in construction.
- UnknownInitialization: meaning do not know an object is fully initialized or not.

The initialization system is introduced in the "freedom before commitment" paper [29] which defines UnknownInitialization as the super type of UnderInitialization and Initialized.

3.1.2 Abstract State

PICO supports abstract state, which provides the ability to explicitly exclude some fields from the guarantee of the transitive immutability. For example, when designing a `Clock` class, we wish to make the UI of the clock, the timezone and date-format immutable, except that the date should be mutable (see listing 3.1).

```java
@Immutable
class Clock {
    @Mutable Date currentDate;
    UI clockUI;
    TimeZone zone;
    Format dateFormat;
    Clock(...) {
        ...
    }
}
```

Listing 3.1: Clock class

By employing the idea of abstract state, only the fields that are in the abstract state are protected by the immutability type system. User can exclude particular fields from the abstract state by explicitly declaring it as `@Mutable` or `@Assignable`. A mutable field cannot be in the abstract state as mutating the field would break the transitive immutability guarantee claimed by this immutable class. An assignable field cannot be in the abstract state as re-assigning the field is essentially mutating one component of this immutable class.

Fields that are commonly considered to not reside in the abstract state are the cache fields (used only for efficient caching, invisible to the clients), the debugging option field (used only for testing) or logging option field.

Table 3.1 lists all combinations of mutability and assignability qualifiers. ✓indicates it is in the abstract state, while ✗is the opposite (the table is the same as the one in thesis [31]).

Assignability \ Mutability	Mutable	Readonly	RDM	Immutable
Assignable	✗	✗	✗	✗
RDA	✗	✗	✓	✓
final	✗	✗	✓	✓

Table 3.1: Combinations of Assignability and Mutability

First of all, declaring a field `@Assignable` exclude that field from the abstract state regardless the mutability qualifiers on them. Second, mutable fields are not in the abstract state, regardless the assignability qualifiers on them. Third, readonly fields are not in the abstract state, as we do not know if it has aliases which could mutate the object elsewhere. For the remaining cases, the fields are in the abstract state.

3.1.2.1 Annotate more to get unsafe behaviors

We obey the principle "annotate more to get unsafe behaviors". That's why we make `@RDM` and `@RDA` as the defaults for fields, so an instance field of an immutable class is by default residing in the abstract state without more actions from the developers. Only when the developer writes annotation explicitly, can the field be excluded. For an example below, it declares an immutable class with a mutable field without explicitly annotating that field as mutable. We are not sure about the users' intent, do they want to achieve full, transitive immutability for `A` or they want to exclude field `b` but forget to annotate it. We would report an `implicit.shallow.immutable` error for this case. Users can address this error by explicitly annotating it (if they want to exclude this field) or changing the mutability of class `B` to `@RDM` or `@Immutable` (if they do not want to exclude this field).

```
@Immutable
class A {
    B b; // did not explicitly exclude field b from the abstract state
}
@Mutable
class B {}
```

We call this kind of field, namely `B b` in the above example, as implicitly mutable field in contrast to explicitly mutable field.

3.1.2.2 Implicitly mutable field in a mutable class

Similar problem could still happen even the implicitly mutable field is declared inside a mutable class, simply because the mutable object could be referenced by a `readonly` reference. By viewpoint adaptation, `a.b` in the below code is mutable and thus mutating it would be allowed. However, for an `immutable` receiver, we would report an `implicit.shallow.immutable` error, for a `readonly` receiver, we should do the same thing for consistency. In order to reach the symmetry in design, we will report an `illegal.field.write` error at each call sites (e.g., line 5 in the below code). User can explicitly annotate the field or change the mutability of the class of that field to address this error.

```
1 @Mutable
2 class A {
3     B b;
4     void static test(@Readonly A a) {
5         a.b.f = 2;
6     }
7 }
8 @Mutable
9 class B {int f = 1;}
```

While table 3.1 shows when mutable or assignable fields are out of the abstract state, table 3.2 shows that users still have to explicitly mark them as mutable and assignable, otherwise errors will be reported.

Field Type Receiver Type	Explicitly Mutable	Implicitly Mutable	(Explicitly) Assignable
Readonly	ok	error at call sites	ok except for one case
Immutable	ok	error at field declaration	ok

Table 3.2: Differences between explicit and implicit annotations

This table describes one kind of field access, `a.f`, where `a` is readonly or immutable and the declaration of `f` is mutable or assignable.

- If `f` is explicitly marked as mutable, PICO allows to mutate `a.f`.

- If `f` is implicitly mutable and `a` is readonly, mutating `a.f` will lead to an `illegal.field.write` error. Note that "implicitly mutable" is not the type of `f`, it just means that `f` is not annotated and the mutability class declaration of `f` is `@Mutable`. In order to report an error in this case, `f` actually defaults to be `@RDM`, so that `a.f` will become readonly after viewpoint adaptation.

- Case that `f` is implicitly mutable and `a` is immutable, cannot happen. Because PICO will issue an `implicit.shallow.immutable` error on the line where such an `f` is declared. Because both RDM and immutable classes can be instantiated to be immutable, we issue this error message at these two classes when fields are implicitly mutable.

- Last, there is no "implicitly assignable" field, because fields are by default RDA, unless users explicitly annotate them to be final or assignable. When the field is assignable and receiver is immutable or readonly, re-assignment to that field is generally allowed. We just exclude one case from this scenario, if the receiver is readonly and the field is not only assignable but also RDM, the re-assignment is disallowed[1].

3.1.3 Transitive Immutability

PICO supports transitive immutability. As the name suggests, transitive immutability guarantees that all fields reachable from an immutable object are immutable. The opposite of transitive immutability is called as shallow immutability which only protects the direct fields from being mutated. Transitive immutability can provide stronger guarantee of immutability to the system.

However, as we introduced the notion of "abstract state", the definition for transitive immutability should be slightly changed. In this thesis, transitive immutability indicates that all objects in the abstract state reachable from the root object are immutable. See listing 3.2 for an example. After writing `@Mutable` on line 8, field `c1` is excluded from the immutability abstract state, thus `a.b.c1` is mutable but `a.b.c2` is immutable.

```
1 @Immutable
2 class A {
3     @ReceiverDependentMutable B b;
4 }
5
```

[1]Detailed discussion of this interesting type rule can found at section 3.1.4 in this thesis or section 4.8.16 in thesis [31]

```
@ReceiverDependentMutable
class B {
    @Mutable C c1;
    @ReceiverDependentMutable C c2;
}

@ReceiverDependentMutable
class C {}
```

Listing 3.2: Transitive Immutability Example

3.1.4 Receiver-Context-Sensitivity

PICO supports two kinds of context sensitivities. The first one discussed in this subsection is called as receiver-context-sensitivity. It essentially means the declared member could be resolved to different types when using different receivers for accessing, hence, sensitive to the receiver type.

Adding receiver-sensitivity makes the type system more expressive and the tool more flexible. Considering a `Scoresheet` implementation. At first, we are still counting and collecting the scores, so we make the `records` field as mutable. However, after done collecting, we wish to keep the integrity of the results. We can either declare a new class `Scoresheet2` with an immutable field, store the final results in that field or we can make that field sensitive to the receiver type. The second class implementation changes the mutability qualifier to `RDM` and the `records` field would become immutable if we have an immutable scoresheet instance. This avoids duplication.

```
class Scoresheet {@Mutable HashMap<Student, Marks> records;}
class Scoresheet {
    @ReceiverDependentMutable HashMap<Student, Marks> records;
}
```

Type qualifier `RDM` and viewpoint adaptation are used to encode the receiver-context-sensitivity. We define our viewpoint adaptation operator $\rhd$ as below.

$$\rhd :: Q \times Q \rightarrow Q$$

$$\text{Readonly} \rhd \text{RDM} = \text{Lost} \tag{3.1.1}$$

$$q \rhd \text{RDM} = q \qquad q \neq \text{Reaodnly} \tag{3.1.2}$$

$$q \rhd q' = q' \qquad \text{otherwise} \tag{3.1.3}$$

First, $\triangleright$ accepts two mutability type qualifiers as input and yields a mutability type. The first type qualifier given is the receiver type and the second type qualifier is the member type. The first formula shows if using `Readonly` as the receiver type and `RDM` as the member type, the output type is `Lost`, which means this mutability type cannot be expressed. `Lost` type is incompatible with `Mutable` and `Immutable`, thus preventing the illegal update. Take the below code for example.

```java
1 @Immutable
2 class AssignableRDM {
3     @Assignable @ReceiverDependentMutable Date d;
4 
5     static void test(@Readonly AssignableRDM ar) {
6         ar.d = new @Mutable Date();
7     }
8 }
```

An immutable class with a field declared with `@Assignable, @RDM`. This means the field is immutable but can be re-assigned. On line 6, we have a readonly receiver `ar`, what's the type of `ar.d`? Suppose do the viewpoint adaptation without the first rule, rule 2 will be applied and `ar.d` is `@Readonly` which can be assigned with a `@Mutable` object. This is against our previous observation that the field is immutable. After adding rule 1, we will have `Lost` type variable on the left-hand-side to prevent the illegal assignment.

The second formula says anything except Readonly "looks at" RDM gives back itself. The third formula says in all other conditions, the output type does not depend on the receiver type. A short example is given below.

```java
@Immutable
class C {
    @Mutable Object f;
    @ReceiverDependentMutable Object f2;

    static void test(@Immutable C c) {
        c.f  // --> @Immutable |> @Mutable = @Mutable
        c.f2 // --> @Immutable |> @RDM     = @Immutable
    }
}
```

Not only does receiver-context-sensitivity apply to field accesses but it also applies to the following places:

- Method Invocation: Similar to field access. The receiver is the caller and the declared receiver type, parameter types and return type are the member types which will be adapted to the receiver type.

- Object Creation: Similar to method invocation. Though there is no receiver when constructing a new object, we take `@A` as the mutability type of the receiver if user writes new operation like this `new @A Object();`. The mutability declared on the constructor is the member type.

- Instantiation of Type Parameters: For a generic class instantiation like this `@A c<@B> = new @A c<>();`, `@A` is the receiver mutability type and the member type is the declared upper/lower bound of the type parameters. `@B` should be within the adapted upper and lower bounds.

3.1.5 Assignment-Context-Sensitivity

PICO also supports assignment-context-sensitivity, which is inspired by Java's parametric polymorphism. The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. Take the below code for example.

```java
class Solution {
    public static <T> T foo(T t) {
        return t;
    }

    static void test() {
        String s = Solution.foo("123");
        String i = Solution.foo(123); // error
    }
}
```

In this assignment context, the left-hand-side target type is used as the required type for the right-hand-side method call. Just like type variable `T`, we introduce `@PolyMutable` to add polymorphism to our immutability type system. This is particularly useful for the factory design pattern — both factory methods in the below example are capable of producing immutable and mutable objects.

```java
@ReceiverDependentMutable
class PolyMutableTest {
    @Mutable MutableBox mb;
    ImmutableBox imb;
    RDMBox rmb;

    PolyMutableTest(MutableBox mb, ImmutableBox imb,
                    @ReceiverDependentMutable RDMBox rmb) {
        this.imb = imb;
        this.mb = mb;
        this.rmb = rmb;
    }

    @PolyMutable Object factory() {
        return new @PolyMutable Object();
    }

    @PolyMutable PolyMutableTest factory2() {
        return new @PolyMutable PolyMutableTest(new MutableBox(), new
    ImmutableBox(),
                new @PolyMutable RDMBox());
    }

     void test() {
        @Immutable Object o = factory();
        @Mutable Object o2 = factory();

        @Immutable PolyMutableTest pt1 =  factory2();
        @Mutable PolyMutableTest pt2 =  factory2();
    }
}

@Mutable
class MutableBox{}
@ReceiverDependentMutable
class RDMBox{}
@Immutable
class ImmutableBox {}
```

`@PolyMutable` is only used on method parameters, method return, receiver and object creation. Assignment-context-sensitivity refinement only happens if there is a method invocation (that is, this polymorphism qualifier is not applied to class type parameters).

3.1.6 Defaulting Mechanism

Default mechanisms are used when some types are not annotated explicitly. Good defaults can reduce users' annotation burden and still achieve safeness and compatibility towards the old unannotated program. Different qualifiers' defaulting rules are described in section 3.1.1, but we have not discussed the priority of applying them if there is a conflict. There are several defaulting mechanisms provided in EISOP Checker Framework manual [10] chapter 31.5, and they are applied in order. For example, an enum class is first a class type and second an enum type. It makes sense to default it to be `@Mutable` and `@Immutable` according to the rules described. I will talk about the priority of applying different defaults in this section.

3.1.6.1 Mutability defaults

First of all, type qualifiers can be applied to type elements (class, interface and enum) and type uses. For classes and interfaces, the default type is `@Mutable`. For enum class, the default type is `@Immutable`. For type uses, defaults are applied based on their actual types and type use locations. We determine the default for a type use by applying these two rules in order: first check the declared type element of this type use, then check the type use location of the type use.

Based on Type Element Some types can only be annotated with `@Immutable`, these types are also called as implicit types. Explicitly annotating it to a different type will raise a `type.invalid.annotation.on.use` error. These types include, all primitive types, `Enum`, `String`, `Double`, `Boolean`, `Byte`, `Character`, `Float`, `Integer`, `Long`, `Short`, `Number`, `BigDecimal`, `BigInteger` and all of them are implicitly annotated to be `@Immutable`. Note here `Enum` refers to the enum constant rather than the enum class. These implicit types are specified by the `@UpperBoundFor` meta-annotation provided by Checker Framework.

Based on Type Use Locations For types that are not considered as implicit types, the type use location matters.

Field: If the type of the field is class or interface, the default type for the field is the mutability type for that class or interface; If the type of the field is a type variable, the default type is `@Sub`. For example, in the below example, each field tries to look up its class declaration and use that as the default type. Type variable does not have a class declaration, so we default it to be `@Sub` which means waiting to be substituted.

```java
class Demo<T> {
    T f; // default mutability type is @Sub
    MutableBox mb; // default mutability type is @Mutable
    RDMBox rdmbox; // default mutability type is @RDM
}
```

```

@Mutable
class MutableBox{}

@ReceiverDependentMutable
class RDMBox{}
```

Local Variable: Always default to `@Readonly` and is subject to flow refinement.

Type Parameter: Having an annotation before a type parameter, like this, `class<@A T extends Object>`, is a way of annotating the lower bound of the type parameter (the reason is that we cannot use `super` when declaring a type parameter). The default type for the type parameter (lower bound) is `@Bottom NullType`.

Array: Array is special, because it's not like class or interface which we could declare the mutability types on it. By default, arrays are declared with `@RDM` mutability type.

Constructor: Constructor shares the same mutability type as the class declaration.

Other type use locations: Other type use locations, such as, object creation, typecast, upper bounds, parameters, etc, have same defaulting rules. First look up the type declaration of these types, (1) if it's `@Mutable, @Immutable`, their default types stay the same; (2) if it's `@RDM`, their default types are `@Mutable`.

3.1.6.2 Initialization defaults

Most of time, objects or `this` reference have `@Initialized` type. In the construction phase, `this` reference is considered as `@UnderInitialization`.

3.1.6.3 Assignability defaults

Instance fields are default to be `@RDA` while static fields are default to be `@Assignable`. Our assignability qualifiers (`@Assignable, @RDA`) are only used on fields.

3.2 Improvements

Above is the up-to-date review of PICO's type system. In this section, we list some improvements we contribute to this type system.

3.2.1 Restricting Type-use Locations of Qualifiers

Feature explained It would be good to have a declared way of restricting the type-use locations of certain qualifiers.

In type-checking, having this feature could reduce boilerplate code because we would not need to write corresponding type rules or well-formedness rules to forbid such misuses. Javac does offer one meta-annotation `@Target({})` to constraint the use of an annotation, but it's very coarse-grained. For example, `Lower_Bound` is not one of the element types so we cannot forbid certain qualifiers from being written on `Lower_Bound` by using the `@Target` meta-annotation. Checker Framework offers another meta-annotation `@TargetLocations()` which accepts `TypeUseLocation` as values, which covers more AST locations. However, this meta-annotation was not enforced by Checker Framework.

Declaring type-use locations of qualifiers also helps our constraint based type-inference system. The old way of preventing the system from inferring certain qualifiers at a location is to extensively add inequality constraints in the visitor methods, which is not that elegant. We wish to a have a mechanism that once we declare the properties of qualifiers then the system takes care of generating constraints at each location.

Approach We pick up the inactive meta-annotation `@TargetLocations` and actually enforce it. This enables a type system designer to permit a qualifier to be applied only in certain locations. By saying "cannot apply", we mean that the qualifier cannot be explicitly written nor be implicitly inferred/computed at some locations. For example, the bottom qualifier in the Signedness type system is defined as below:

```java
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@TargetLocations({
    TypeUseLocation.LOWER_BOUND,
    TypeUseLocation.UPPER_BOUND,
})
@SubtypeOf({SignedPositiveFromUnsigned.class})
public @interface SignednessBottom {}
```

This requires `SignednessBottom` to only appear in the wildcard or type parameter declaration. Note we use `@Target` meta-annotation together with `@TargetLocations`, because `@Target` is enforced by Javac and there is no way bypassing it. In Checker Framework we provide a flag `-AignoreTargetLocations` to bypass the `TargetLocations` checks.

In all, the goal of the check focuses on the well-formedness of types, like `Bottom` is often an invalid type for local variables. If developers want an even more fine-grained approach, for example limiting qualifier uses in places like object creation or typecasting, they still need to translate these constraints into type rules and implement it in the visitor method.

3.2.2 Warn Redundant Annotations

Problem explained Sometimes users write annotations that are the same as the default ones would be applied. A fully annotated Java class is clear in its semantics but poor in its readability.

Approach We provide a mechanism to suggest users not writing some qualifiers explicitly in the source code because they are the defaults for these locations, writing them down would be redundant. We make this check optional, only when users provide the `-AwarnRedundantAnnotations` flag, the restriction is enforced and `redundant.anno` warnings could be issued. Without supplying the flag, the check is by default disabled because performing this check on every `variabletree` is time-consuming.

3.2.3 Apply Flow-Refinement for Receivers when Computing the Type of LHS

Problem explained Receivers having no flow refinement has been in a long going issue in Checker Framework. This is a general issue in Checker Framework but let's examine an example in the immutability type system (the type lattice is at figure 3.1).

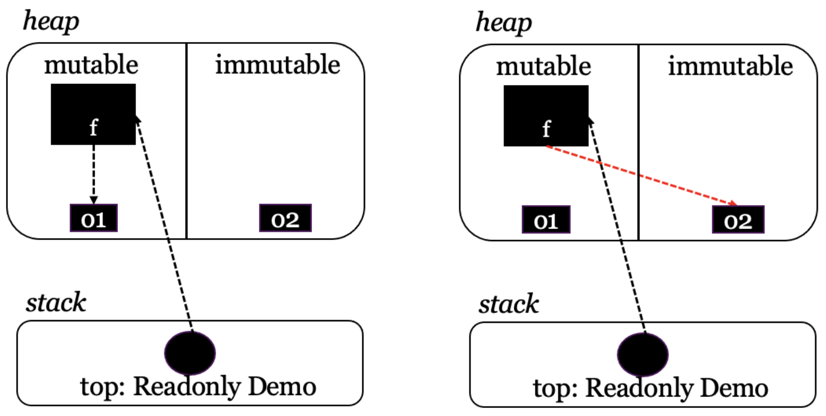
```java
class Demo {
    @RDM Object f;

    static void adaptation(@Mutable Demo p) {
        @Readonly Demo top = p;
        // expects no error
        top.f = new @Mutable Object();
        // expects an error
        top.f = new @Immutable Object();
    }

    static void test() {
        @Readonly Object o = new @Mutable Object();
        o = new @Immutable Object();
    }
}
```
Listing 3.3: Listing to illustrate the need for having refinement for receivers

A type system and a piece of code is provided as above, where `Readonly, Bottom` are the top and bottom qualifiers in the type system, `Mutable, Immutable` are qualifiers in the middle, and `RDM` stands for the receiver-dependent qualifiers (used for viewpoint adaptation). The code defines a class `Demo` with an `RDM` field, and in the static method assigns a type `Mutable Demo` instance to a `Readonly Demo` variable, and then performs two assignment operations on the the field of `top`.

Normally when computing the annotated type of the left-hand-side (lhs in short) tree, the flow-refinement is disabled. The motivation is that variable on the lhs will be refined by the right-hand-side (rhs in short) value in the assignment. The behavior is illustrated by the static method `test` in the above code. In the method, `o` is first refined to an `Mutable` instance and then point to an `Immutable` instance. If we have flow-refinement enabled when computing the type of `o`, then assigning `b` to `o` would be forbidden, which does not make sense. Why? This is the same as in Java, an object can point to a string value and then point to a integer instance. Here, `Readonly` reference can point to an `Mutable` instance and can be re-assigned with a `Immutable` instance.

So, forbidding flow-refinement on the lhs reference seems the right thing to do, why are we enabling it under certain conditions? Things are a bit different when viewpoint adaptation comes into play. Let's examine the code in the `adaptation` method. `top` reference points to a `Mutable Demo` variable after line 5. Suppose we still give no flow refinement when computing the type of the lhs reference and do not use the new `@Lost` type in type checking, the assignment on line 9 would be accepted. Statically, `top` is annotated by `@Readonly` and field `f` is receiver-dependent, `top.f` is then in `@Readonly` type which is a supertype of `Immutable` (the type of the rhs instance). But at runtime, `top` actually points to an `@Mutable Demo` instance, and `top.f` should be typed to be `@Mutable`. Now the mutable instance captures an immutable instance and breaks the type safety.

Having the flow-refinement only for the receiver would solve this issue. The receiver of `top.f`, which is `top` would have type `@Mutable`, and the field access would be `@Mutable` as well. Thus prevents the illegal assignment.



(a) Memory layout before line 9 (b) Memory layout after line 9

Figure 3.2: Memory diagram for listing 3.3

Figure 3.2 gives two memory diagrams before and after the assignment at lint 9. Figure (a) has a `@Readonly top` reference pointing at a mutable object on the heap, and that mutable object contains a mutable reference `f` pointing to a mutable `o1`. After line 9, the reference `f` points to `o2`, the immutable object, which violates the immutability guarantee we want to enforce on the heap. Only do we refine `top` to become `@Mutable`, reference `f` can become `@Mutable` (by viewpoint adaptation) and reject the illegal assignment at line 9 in the typecheck phase.

Approach `GenericAnnotatedTypeFactory` introduces a boolean field called `computingAnnotatedTypeMirrorOfLHS` to solve this issue. However, the field is private and making it protected (so that the derived class could use) could weaken the abstraction of that class. We solve this issue by resetting the `useFlow` to the original value when visiting the member selection tree.

3.2.4 Adding the Lost Type Qualifier to the System

The above mentioned approach in section 3.2.3 does not solve the type safety issue entirely. If we change the previous code a little bit, another problem would arise. Take the code below for instance.

```
static void adaptation(@Readonly Demo top) {
    top.f = new @Immutable Object();
}

adaptation(new @Mutable Demo());
```

The assignment would be accepted, as `top.f` has type `@Readonly` and the rhs is typed to be `@Immutable`. However, an invocation of the method with an mutable method argument would break the type safety of the program. The assignment again makes a mutable instance capture an immutable instance.

Approach Our improvement is to add `@Lost` qualifier to the system and `@Readonly` ▷ `@RDM` yields the `@Lost` qualifier. `@Lost` qualifier is incompatible with `@Mutable` and `@Immutable` types and forbids these illegal assignments. This rule is discussed in section 3.1.4.

3.2.5 Improve Typecast Logic

Feature explained The newly implemented typecast logic cleanly separates upcast, downcast and incomparable cast when comparing the casting expression and the target expression. Note that in these optional type systems, casting is determined from two different aspects. For example, casting a `@Immutable Object` to `@Readonly String` is a downcast in terms of the Java types (`Object` to `String`) but a upcast from the perspective of type qualifier (from `@Immutable` to `@Readonly`). If not explicitly mentioned, upcast, downcast refers to the type qualifier perspectives.

Casting a type qualifier to its supertype (upcast) is always allowed. If it is not a upcast, we further determine if it is a safe downcast (also called as statically-verifiable downcast). Some downcasts can be statically verified and should not issue a warning or error. Suppose we are casting a `@Top base` variable to `@Bot derived` type, both the pluggable types and Java types are downcasting, and suppose derived class can only be annotated with `Bot`. We believe this kinds of downcast can be statically verified in Checker Framework. Case 1: if base is really a derived instance and this instance could only be annotated with `@Bot`, it's okay that we cast from `@Top` to `@Bot`; case 2: if base is not a derived instance, and we can rely on javac or java-runtime to report a inconvertible types or ClassCastException. On the contrary, if the downcast cannot be statically verified, Checker Framework will report a warning to users. Last, if it's neither upcast nor downcast, it must be an incomparable cast, which is generally regarded as illegal. In this case, we report an error demanding developers to re-check their code. Specific type system can override this incomparable casting procedure to allow certain incomparable cast.

Below is a table summarizing the error/warning types for different cases.

Java Type / Qualifier	Up	Down	Same
Up/Same	ok		
Down	cast.warning	some can be verified (bounds == castType.getAnnotations()),other conditions with a cast.warning	cast.warning
Incomparable	cast.error		

Figure 3.3: Typecast errors/warnings

Casting involved with type variables will become more complicated. First of all, in Checker Framework, we should not be too worried about casting an object to a type variable or casting a type variable `U` to type variable `T`. In both cases, javac will report an "unchecked.cast" warning indicating lack of static guarantee on these castings. We should be more careful when casting a variable in type `T` to type variable `T`. Javac cannot identify any difference in this case, but from the type qualifier perspective, the two type variables with the same name could have different type qualifiers, for example, see line 4 of the code below.

```java
1  class Inner<T extends @UnknownSignedness Object> {
2      T bar(@Signed T p) {
3          // :: warning: (cast.unsafe)
4          _ = (T) p;
5          return x;
6      }
7      void foo(Inner<@SignednessGlb Integer> s, @Signed Integer local) {
8          @SignednessGlb Integer x = s.bar(local);
9      }
10 }
```

The typecast in method `bar` looks like an upcast (in terms of the qualifier, `p` is `signed` and `T` extends `UnknownSignedness`), but it could be a downcast in invocation (See method `foo` below for an example). We should report a downcast warning if there is one because javac does not warn when casting a variable from type T to T. In method `foo`, we passed in an @Signed integer and the method signature after substitution is @SignednessGlb Integer bar(@Signed Object). This makes the typecast discussed in method bar a downcast.

Contribution In implementation, (1) we identify the statically-verifiable downcast cases which now does not report any error; (2) makes incompatible casts throw errors instead of warnings; (3) add additional checks when casting two type variables with the same name.

3.2.6 Improvements to Transitive Immutability

Problem explained PICO guarantees that an immutable object/reference or a readonly reference cannot be mutated unless users have explicitly excluded the fields of that object or reference out of the abstract state. With that mindset, this code will issue immutability errors to notify users. Users either annotate field f to make that field out of the abstract state, meaning this field is out of the immutability contract and could be mutated, or change the class mutability of A to @Immutable or @RDM, meaning that the A could be instantiated to be immutable as well.

```java
class A {}
@Immutable
class B {
    /*@Mutable*/ A f;
}
```

Same logic applies to @Readonly references. According to the definition of @Readonly: the object it points to could be mutated by other aliases, but the readonly reference cannot be used to mutate the object.

```java
/*@Mutable*/class B {
    MutableBox f;
    static void test(@Readonly B b) {
        // :: error: (illegal.field.write)
        b.f.field = 1;
    }
}
/*@Mutable*/class MutableBox {
    int field = 0;
}
```

So, in the above code, the attempt to mutate the object through that readonly reference fails. Wait, but if we use the viewpoint adaptation rule, b.f yields @Mutable type, then how is that field update disallowed? This is a tricky point in PICO, because we would force PICO to annotate a default mutable field to `@RDM` in an `@RDM` class or a `@Mutable` class. So, the default mutable field `f` is annotated by PICO to be `@RDM`. Then doing the viewpoint adaptation yields readonly type and the assignment on line 5 is forbidden. The previous thesis [30] gives a thorough discussion on this design and other alternatives in section 3.6.2. However, another problem arises when replacing `@Mutable` with `@RDM` in an `@RDM` class.

```
1  @ReceiverDependentMutable
2  class RDMBox {
3      MutableBox mb;
4
5      void test(@Immutable RDMBox this) {
6          this.mb.field = 1;
7      }
8  }
9  /*@Mutable*/class MutableBox {
10     int field = 0;
11 }
```

An `RDM` class can be instantiated to be immutable or mutable. Then, on line 6 of the above example, `this.mb` resolves to be immutable if `mb` is annotated to be `RDM` by PICO. However, a mutable class can only be instantiated to be mutable according to the definition, and now it has a immutable reference, which breaks the type safety.

Approach The design we adopted to achieve transitive immutability and fix this bug is illustrated in table 3.2. While the earlier section already introduces the design, this section more focuses on what problems we ran into and our reasoning behind the solution.

3.2.7 Other Improvements

Other improvements worth mentioning are listed here.

3.2.7.1 Making defaults explicit in source code

Added a flag in Checker Framework Inference to make all default qualifiers explicit. This could make the source code more cluttered but it's definitely useful for debugging. More, this also solves several test cases failures as CF and CFI now do not agree on defaults even for the same location and type. More specifically, CFI decides the inferred qualifier is the same as the default at one particular slot and thus not writing it in the source code, but in the typecheck mode, the default qualifier CF computed for this slot is different and thus typecheck fails. User needs to override the `makeDefaultsExplicit` method in `CFInferenceTest` to enable this feature.

3.2.7.2 Constructor

Previously, for new class tree like `new @Slot1 ArrayList<String>();` the default type for slot1 was derived by visiting the identifier tree, not the new class tree. By overriding this method, the default type for slot1 now is derived by visiting the new class tree, which is a more precise context for annotating. I also override the `constructorFromUse` methods in both PICO and PICO-infer AnnotatedTypeFactory, to apply a correct default for the constructor return type.

3.2.7.3 Refactoring

Refactoring the old immutability project requires a lot of effort. The first stage is to successfully compile PICO-Typechecker and PICO-Inference. Like said earlier, PICO was based on a relatively old version of Checker Framework and on average 1000 commits are merged into Checker Framework per year and a lot of new APIs are not backward compatible. PICO-TypeChecker had more than 100+ compilation errors initially. Based on the changelog, information in the pull requests, the easier ones can be solved by API renaming or changing the method signature. Harder ones are usually caused by data abstraction. Some fields are internal and made private, so when overriding the basetypechecker, we do not have the access to some fields. Also, copying the parent method completely to the child class and changing just a few lines of code is not a recommended coding practice. In this way, we usually have to propose new solutions to achieve similar functionalities.

The second stage focuses on the semantics of the project, to make it pass all test cases. Problems include: (1) error/warning messages in the test files are outdated and should be updated or removed; (2) problems rooted in Checker Framework or Checker Framework Inference and should be fixed there; (3) some type rules and well-formedness rules are missing or incorrect in the system; (4) memory issues in the project, caused by insufficient heap and circular referencing; (5) issues in corner cases, such as enums, intersection type.

Right now, we have fixed all bugs in PICO typecheck part and we still need to make this project as the downstream test in the Checker Framework building pipeline. Unfortunately, we have not finished refactoring PICO type inference and this is left as the future work.

3.3 Formalization

This section formalizes the core part of PICO: we exclude features like generics and method overriding. The language details can be seen in the syntax.

Note that previous theses both provide formalization to PICO (section 4.9.1 in thesis [31] and section 3 in thesis [30]), my contribution lies in:

- The old formalization is not up to date with the current implementation of PICO. We updated the polymorphism and viewpoint adapted part.
- The old formalization is not complete. We added more well-formedness rules, type-cast expressions, subtyping & subclassing rules and enables the check for assignability.
- The old formalization is complicated and lack of explanation. We re-organize the preconditions and extract common helper functions to make syntax and rules neat. Detailed explanations are provided as well.
- Provided the operation semantics to describe the execution of the program.

3.3.1 Helper Functions

- **bound**$(C) = q$. This returns the mutability declaration on class C.
- **checkSuperCall**$(\overline{x}, C)$ accepts a list of arguments $\overline{x}$ and a class C. Returns true if the type of the list arguments match the adapted parameter type of the constructor of class D (which is the parent class of class C).
- **fType**(C, f) returns the declared type of field f.
- **mSig**(C, m) returns the declared signature of method m in class C.
- **conSig**(C) returns the declared signature of constructor in class C.
- **parentClass**(C) returns the direct superclass of class C.
- **isInConstructor**(σ) given a σ stack frame, returns if the current calling context is a constructor or not.
- **fields**(C) returns the list of fields given a class identifier C.
- **isAssignableField**(C, f) returns if the field f is assignable declared in class C.
- **DeclBound**(C) returns the declared mutability bound of a class.

3.3.2 Syntax

The syntax is based on the previous thesis [31] with some modifications.

$$
\begin{aligned}
P &::= \overline{cd}\ s \\
cd &::= q \text{ class } C \text{ extends } D\ \{\overline{fd}\ kd\ \overline{md}\} \\
fd &::= (a\ q\ C)\ f \\
md &::= (T\ C)\ \mathrm{m}((T'\ C')\ \mathit{this},\ \overline{(T''\ C'')\ x})\{s;\ \text{return } z;\} \\
kd &::= q\ C\ (\overline{(T\ C)\ x},\ \overline{(T'\ C')\ y})\ \{\text{super}(\overline{x});\ \overline{\mathit{this}.f = y}\} \\
e &::= x \mid x.f \mid \mathit{null} \\
s &::= x = e \mid x.f = y \mid x = y.m(\overline{z}) \mid x = \text{new } (q\ C(\overline{y})) \mid x = (T\ C)\ y \mid s; s \\
T &::= k\ q \\
k &::= \text{initialized} \mid \text{underinitialization} \\
&\quad\ \mid \text{unknowninitialization} \mid \text{fbcbottom} \\
q &::= \text{readonly} \mid \text{mutable} \mid \text{polymutable} \\
&\quad\ \mid \text{rdm} \mid \text{immutable} \mid \text{bottom} \\
a &::= \text{assignable} \mid \text{rda} \mid \text{final}
\end{aligned}
$$

x, y, z, p	variable identifiers
$f,\ m$	field, method identifiers
C, D	class identifiers

A program consists of a set of classes and a main statement $\overline{s}$. A class cd is a non-generic class C extending class D with a set of fields $\overline{fd}$, methods $\overline{md}$ and one constructor kd. More, the class is annotated by a mutability qualifier q. A field fd is composed of a type and a field identifier f. A non-static non-generic method md has return type $(T\ C)$, a receiver parameter in type $(T'\ C')$ and a sequence of parameters in type $(T''\ C'')$. The method body is a set of statements followed by a return statement. A constructor declaration kd is similar to the method declaration. We have two sets of parameters x and y, one goes to the super call, one is used to initialize all fields of C. Constructor does not have a return type, but the constructor identifier is annotated with a mutability qualifier indicating the mutability of the initialized object constructed.

An expression is without side-effect and is either a variable or a field access. A statement is perhaps a (1) variable assignment, (2) field update, (3) method invocation, (4) new operation or (5) a conjunction of two statements. k, q, a are qualifiers from the initialization, mutability and assignability dimension. T is a type composed of k, q.

3.3.3 Subtype & Subclassing rules

In our type system, `T` has two dimensions, one is the initialization dimension, another is the mutability dimension. The lattice for the two dimensions can be seen at figure 3.1 and section 3.8.1 in the manual [10].

$$\frac{}{\Gamma \vdash T <: T} \quad (\text{ST-Reflexivity}) \qquad \frac{T <: T' \qquad T' <: T''}{\Gamma \vdash T <: T''} \quad (\text{ST-Transitivity})$$

$$\frac{T <: T' \qquad C <: D}{\Gamma \vdash (T\ C) <: (T'\ D)} \quad (\text{ST-SubType}) \qquad \frac{T <: T' \qquad \mathit{Lost} \notin T'}{T <:_s T'} \quad (\text{ST-Strict})$$

$$\frac{C \sqsubseteq C' \qquad C' \sqsubseteq D}{C \sqsubseteq D} \quad (\text{SC-Transitivity}) \qquad \frac{}{\Gamma \vdash C \sqsubseteq C} \quad (\text{SC-Reflexivty})$$

$$\frac{q \text{ class } C \text{ extends } D\ \{\overline{fd}\ kd\ \overline{md}\}}{C \sqsubseteq D} \quad (\text{SC-Extends})$$

3.3.4 Well-Formedness Rules

$$\frac{q \rhd \text{bound}(C) <: q \qquad q \neq \text{Bottom}}{(q\ C)\ \text{ok}} \quad (\text{WF-TypeUse})$$

$$\frac{q \neq \text{PolyMutable}}{(a\ q\ C)\ f\ \text{ok}} \quad (\text{WF-Field})$$

$$\frac{\begin{array}{c}\Gamma = \{\mathit{this} \to (\text{underinitialization } q\ C), \overline{x} \to (T\ C), \overline{y} \to (T'\ C')\} \\ \text{checkSuperCall}(\overline{x}, C)\ \text{ok} \qquad \Gamma \vdash \overline{T'\ C'} <: \overline{\text{FType}(C, f)}) \\ (T\ C),\ (T'\ C')\ \text{ok} \qquad q \rhd \text{bound(C)} = q \quad q \neq \text{Readonly} \quad q \neq \text{PolyMutable}\end{array}}{q\ C\ (\overline{(T\ C)\ x},\ \overline{(T'\ C')\ y})\ \{\text{super}(\overline{x});\ \overline{\mathit{this}.f = y}\}\ \text{ok}} \quad (\text{WF-Cons})$$

$$\frac{\begin{array}{c}\Gamma(\overline{x}) = \overline{k_x q_x\ C_x} \qquad \text{parentClass}(C) = D \qquad \text{conSig}(C) = q_c\ C(\_) \\ \text{conSig}(D) = \_\ D(\overline{(k_d q_d\ D)\ p}) \qquad \overline{k_x q_x\ C_x} <: \overline{k_d(q_c \rhd q_d)\ D}\end{array}}{\text{checkSuperCall}(\overline{x}, C)\ \text{ok}} \quad (\textsc{checkSuperCall})$$

$$\frac{\begin{array}{c}\Gamma = \{\mathit{this} \to (k'q'\ C), \overline{x} \to (k''q''\ C)\} \\ \Gamma \vdash s : \_ \qquad \Gamma \vdash \Gamma(z) <: (kq)\ C \qquad (kq\ C),\ (k'q'\ C'),\ (k''q''\ C'')\ \text{ok}\end{array}}{(kq\ C)\ \text{m}((k'q'\ C')\ \mathit{this},\ \overline{(k''q''\ C'')\ x})\{s;\ \text{return}\ z;\}\ \text{ok}} \quad (\text{WF-}\textsc{Meth})$$

$$\frac{\begin{array}{c}\overline{fd}, kd, \overline{md}\ \text{ok} \qquad q \rhd \text{bound}(D) = q \\ q \neq \text{PolyMutable} \qquad q \neq \text{Bottom} \qquad q \neq \text{Readonly}\end{array}}{q\ \text{class}\ C\ \text{extends}\ D\ \{\overline{fd}\ kd\ \overline{md}\}\ \text{ok}} \quad (\text{WF-}\textsc{Class})$$

$$\frac{\overline{cd}\ \text{ok} \qquad \emptyset \vdash s : N}{\overline{cd}\ s\ \text{ok}} \quad (\text{WFP})$$

Generally the use of mutable types is okay if it satisfies the viewpoint adapted subtype constraint (introduced in thesis [30] at section 3.3.2) and the mutable type is not `@Bottom` (WF-TypeUse). WF-Field adds additional constraint that `@PolyMutable` is not allowed on fields. Note that field is also a type use, so when type checking, the annotated field will be checked by WF-TypeUse and WF-Field.

WF-Cons checks the constructor declaration of a given class. The first line of the preconditions builds up the type environment; the second line first checks if the arguments match the adapted super method declaration and then checks subtype relationship of the remaining arguments and the fields of the class (in the constructor body, we explicitly requires that every field must be initialized and no re-assignment is allowed); Last line requires the well-formedness of the parameters and q cannot be `@Readonly` and `@PolyMutable`. The following checkSuperCall helper function checks the super method invocation.

WF-Meth first builds up the type environment Γ and then checks the statement s in the method body, the subtype relationship between the return expression and the return type and the well-formedness of the types in the method declaration.

For rule WF-Class, the first judgement on the right top is the viewpoint adapted subtype constraint. It essentially requires the mutability of declared class C should be compatible with the parent class D. q can only be `@Immutable`, `@Mutable` or `@RDM`. A program is well-formed if all classes in the program are well-formed and the main statement is well-typed (WFP).

3.3.5 Static Type Checking

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad (\text{T-}\textsc{Var}) \qquad \frac{\Gamma \vdash e : T \qquad T <: \Gamma(x)}{\Gamma \vdash x = e : T} \quad (\text{T-}\textsc{VarAss})$$

$$\frac{\begin{array}{c}\Gamma(x) = k_x q_x\ C_x \qquad \mathrm{FType}(C, f) = \_\ q\ C_f \qquad q' = q_x \rhd q \\ k_x = \mathrm{initialized} \Rightarrow k = \mathrm{initialized} \\ k_x \neq \mathrm{initialized} \Rightarrow k = \mathrm{unknowninitialization}\end{array}}{\Gamma \vdash x.f : kq'\ C_f} \quad (\text{T-}\textsc{FldRead})$$

$$\frac{\begin{array}{c}\Gamma \vdash x :_s (k_x q_x\ C_x),\ y : (k_y q_y\ C_y) \qquad \mathrm{FType}(C, f) = a_f k_f q_f\ C_f \\ k_x = \mathrm{underinitialization} \vee k_y = \mathrm{initialized} \\ (k_x = \mathrm{underinitialization}) \\ \vee (q_x = \mathrm{Mutable} \wedge a_f \neq \mathrm{final}) \vee (a_f = \mathrm{assignable} \wedge (q_x \rhd q_f \neq \mathrm{Lost}))\end{array}}{\Gamma \vdash x.f = y : (k_y q_y\ C_y)} \quad (\text{T-}\textsc{FldAss})$$

$$\frac{\begin{array}{c}\textsc{mSig}(C, m) = (k' q_{ret}\ C')\ \mathrm{m}((k q_{this}\ C)\ \mathit{this},\ \overline{(k'' q''\ C'')\ p}) \\ q_{sub} \neq \mathrm{Bottom} \qquad q_{sub} \neq \mathrm{PolyMutable} \qquad q_{ret} = \mathrm{PolyMutable} \Rightarrow q_{sub} <: q_x \\ q_{this} = \mathrm{PolyMutable} \Rightarrow q_y <: q_{sub} \qquad \mathrm{PolyMutable} \in \overline{q''} \Rightarrow \overline{q_z} <: q_{sub} \\ q_y = \mathrm{Readonly} \Rightarrow q = \mathrm{Lost} \qquad q_y \neq \mathrm{Readonly} \Rightarrow q = q_y\end{array}}{\textsc{mSig2}(q_x, q_y, \overline{q_z}, C, m) = \textsc{mSig}(C, m)[q_{sub}/\mathrm{PolyMutable}][q/\mathrm{RDM}]} \quad (\textsc{mSig2})$$

$$\frac{\begin{array}{c}\Gamma \vdash x :_s (T_x\ C_x) \qquad \Gamma \vdash y : (T_y\ C_y) \qquad \Gamma \vdash \overline{z} : \overline{T_z\ C_z} \\ \textsc{mSig2}(\Gamma(x), \Gamma(y), \Gamma(\overline{z}), C_y, m) = (T\ C)\ \mathrm{m}((T'\ C')\ \mathit{this},\ \overline{(T''\ C'')\ x}) \\ \Gamma \vdash (T\ C) <: (T_x\ C_x) \qquad (T_y\ C_y) <: (T'\ C') \qquad \overline{T_z\ C_z} <: \overline{T''\ C''}\end{array}}{\Gamma \vdash x = y.m(\overline{z}) : (T_y\ C_y)} \quad (\text{T-}\textsc{Call})$$

$$\frac{\begin{array}{c}\Gamma \vdash x : (k_x q_x\ C_x) \qquad \Gamma \vdash y : (k_y q_y\ C_y) \qquad \mathrm{conSig}(C) = q_c C(\overline{k_p q_p\ C_p\ p}) \\ q \neq \mathrm{Readonly} \qquad q = q \rhd q_c \qquad \overline{k_y q_y\ C_y} <: \overline{k_p (q \rhd q_p)\ C_p} \qquad kq\ C <: k_x q_x\ C_x \\ \overline{k_p} = \overline{\mathrm{initialized}} \Rightarrow k = \mathrm{initialized} \qquad \overline{k_p} \neq \overline{\mathrm{initialized}} \Rightarrow k = \mathrm{underinitialization}\end{array}}{\Gamma \vdash x = \mathrm{new}\ q\ C(\overline{y}) : (kq\ C)} \quad (\text{T-}\textsc{New})$$

$$\frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s1; s2} \quad (\text{T-SEQ}) \qquad \frac{\Gamma(x) = T_x\ C_x \qquad T >: T_x}{\Gamma \vdash (T\ C)\ x : (T\ C)} \quad (\text{T-UPCAST})$$

$$\frac{\begin{array}{c}\Gamma(x) = T_x\ C_x \qquad kq <: T_x \qquad C <: C_x \\ q = \text{DeclBound}(C) \qquad \text{DeclBound}(C) \neq \text{RDM}\end{array}}{\Gamma \vdash (kq\ C)\ x : (kq\ C)} \quad (\text{T-DOWNCAST})$$

We simply looks up from the type environment for the type of a variable (T-VAR). For variable assignment, we have to make sure the subtype relationship match (T-VARASS). For field access (T-FLDREAD), the final mutability type is derived by viewpoint adaptation and the final initialization type depends on the initialization type of the receiver.

Rule T-FLDASS looks complicated and it does these things: the first line gathers type information from Γ. The second line is the assignment constraint added by the initialization checker. The third, forth line checks if $x.f$ is assignable or not, which contains three disjunctive clauses. These scenarios are listed below:

- k_x = underinitialization. This handles the construction-phase field update, where immutable and mutable fields are allowed to mutate. As the WF-CONS rule requires no field re-assignment, we do not need to consider final fields re-assignment here.
- (x is mutable) AND ($f \neq$ final). This is the most common condition, where x is mutable.
- (f is assignable) AND ($x.f$ is not Lost). This considers the fields excluded out of the abstract state.

Rule T-CALL checks the subtype relationship for return, parameters and this-parameter, based on the adapted method signature. We get the adapted method signature first by doing the polymorphic qualifier substitution (see line 2-3 in the preconditions of MSIG2) and then perform the viewpoint adaptation(see line 4 in the preconditions of MSIG2).

In MSIG2, we first introduce a placeholder qualifier q_{sub} to substitute possible `PolyMutable` qualifiers in the method signature. If `PolyMutable` is annotated on the return type, then it is substituted by q_{sub} and requires q_{sub} should be the subtype of the actual return type q_x. Similarly, if `PolyMutable` is annotated on the method parameters then it is substituted by q_{sub} and requires q_{sub} to be the supertype of the argument type. Last, methSig[q_{sub}/PolyMutable] denotes using q_{sub} to replace `PolyMutable` in the methSig. The fourth line in the preconditions invents a new qualifier q and will be used to substitute `RDM` in the signature later. We just have to be careful that `Readonly` $\rhd$ `RDM` gives `Lost` instead of `Readonly`.

T-New types the object creation statement. It uses q, the qualifier after the `new` keyword for viewpoint adaptation and then carries out the subtyping check. The last line is the constraint introduced by the initialization checker.

We permit two kinds of statically verifiable casts T-Upcast and T-Downcast. While upcast is always safe, downcast requires the declaration bound of the target type is the same as the mutability qualifier used on the casting type. Also, the declaration bound should not be `RDM`.

3.3.6 Operational Semantics

$$\frac{\begin{array}{c} v \notin \text{dom}(h) \qquad \text{fields(C)} = \overline{f} \qquad q = \text{Immutable} \vee q = \text{Mutable} \\ \text{conSig}(C) = \_C(\overline{\_\ p}) \qquad h' = h[v \to (q\ C, \overline{f \to null_a})] \\ \text{conBody}(C) = s \qquad \sigma_1 = [\text{this} \to v, \overline{p \to \sigma(y)}, \text{true}] \qquad \text{ok}, \sigma_1, h', s \rightsquigarrow \sigma_1, h'', \epsilon \end{array}}{\text{ok},\ x = \text{new}\ (q\ C(\overline{y})), \sigma, h \rightsquigarrow \sigma[x \to v], h'', \epsilon} \quad \text{(R-New1)}$$

$$\frac{\begin{array}{c} v \notin \text{dom}(h) \qquad \text{fields(C)} = \overline{f} \\ h_t(\sigma(x)) = q_x\ C_x \qquad q = \text{PolyMutable} \\ \text{conSig}(C) = \_C(\overline{\_\ p}) \qquad h' = h[v \to (q_x\ C, \overline{f \to null_a})] \\ \text{conBody}(C) = s \qquad \sigma_1 = [\text{this} \to v, \overline{p \to \sigma(y)}, \text{true}] \qquad \text{ok}, \sigma_1, h', s \rightsquigarrow \sigma_1, h'', \epsilon \end{array}}{\text{ok},\ x = \text{new}\ (q\ C(\overline{y})), \sigma, h \rightsquigarrow \sigma[x \to v], h'', \epsilon} \quad \text{(R-New2)}$$

$$\frac{\begin{array}{c} v \notin \text{dom}(h) \qquad \text{fields(C)} = \overline{f} \qquad q = \text{RDM} \\ h_t(\sigma(\text{this})) = q_{this}\ C_{this} \qquad h_t(\sigma(x)) = q_x\ C_x \\ \text{conSig}(C) = \_C(\overline{\_\ p}) \qquad h' = h[v \to ((q_{this} \rhd q)\ C, \overline{f \to null_a})] \\ \text{conBody}(C) = s \qquad \sigma_1 = [\text{this} \to v, \overline{p \to \sigma(y)}, \text{true}] \qquad \text{ok}, \sigma_1, h', s \rightsquigarrow \sigma_1, h'', \epsilon \end{array}}{\text{ok},\ x = \text{new}\ (q\ C(\overline{y})), \sigma, h \rightsquigarrow \sigma[x \to v], h'', \epsilon} \quad \text{(R-New3)}$$

$$\frac{\sigma(x) = v \qquad \sigma(y) = v' \qquad h_t(v) = \text{Mutable}\ C_x \qquad h' = h[v.f = v']}{\text{ok},\ x.f = y, \sigma, h \rightsquigarrow \sigma, h', \text{ok}} \quad \text{(R-FldAss)}$$

$$\frac{\begin{array}{c}\sigma(x) = v \qquad \sigma(y) = v' \\ h_t(v) = \text{Immutable } C_x \qquad h' = h[v.f = v'] \\ \text{isInConstructor}(\sigma) \vee (\neg\text{isInConstructor}(\sigma) \wedge \text{isAssignableField}(C_x, f))\end{array}}{\text{ok, } x.f = y, \sigma, h \rightsquigarrow \sigma, h', \text{ ok}} \quad (\text{R-FldAss2})$$

$$\frac{\begin{array}{c}\sigma(x) = v \qquad h_t(v) = \text{Immutable } C_x \\ \neg\text{isAssignableField}(C_x, f) \qquad \neg\text{isInConstructor}(\sigma)\end{array}}{\text{ok, } x.f = y, \sigma, h \rightsquigarrow \sigma, h, \text{ ImmExc}} \quad (\text{R-FldAssExc})$$

$$\frac{\begin{array}{c}\sigma(x, y, \overline{z}) = v_x, v_y, \overline{v_z} \qquad h_t(v_x) = (q_x\text{\_}) \qquad h_t(v_y) = (q_y C_y) \qquad h_t(\overline{v_z}) = (\overline{q_z C_z}) \\ \text{MBody}(C_y, m) = \{s; \text{return } res\} \\ \sigma_1 = [\text{this} \to v_y, \overline{p} \to \overline{v_z}, res \to null, \text{false}] \qquad \text{ok}, s, \sigma_1, h \rightsquigarrow \sigma', h', \epsilon\end{array}}{\text{ok, } x = y.m(\overline{z}), \sigma, h \rightsquigarrow \sigma[x \to \sigma'(res)], h', \epsilon} \quad (\text{R-Call})$$

$$\frac{\sigma(y) = v \qquad h_t(v) = (q\ C)}{\text{ok, } x = (kq\ C)\ y, \sigma, h, \rightsquigarrow \sigma[x \to v], h, \text{ ok}} \quad (\text{R-Cast})$$

$$\frac{\sigma(y) = v \qquad h_t(v) = (q\ C) \qquad q'\ C' \neq q\ C}{\text{ok, } x = (kq'\ C')\ y, \sigma, h, \rightsquigarrow \sigma, h, \text{ castExc}} \quad (\text{R-CastExc})$$

$$\frac{\begin{array}{c}\text{ok, } s_1, \sigma, h \rightsquigarrow \sigma_1, h_1, \epsilon \\ \epsilon, s_2, \sigma_1, h_1 \rightsquigarrow \sigma_2, h_2, \epsilon'\end{array}}{\text{ok, } s_1; s_2, \sigma, h \rightsquigarrow \sigma_2, h_2, \epsilon'} \quad (\text{R-Seq})$$

$$\frac{\text{eval}_{h,\sigma}(e) = v}{\text{ok, } x = e, \sigma, h \rightsquigarrow \sigma[x \to v], h, \text{ ok}} \quad (\text{R-VarAss})$$

$$\frac{\sigma(x) = v}{\text{eval}_{h,\sigma}(x) = v} \quad (\text{R-Var}) \qquad \frac{\sigma(x) = v \qquad f \in \text{fields}(h_t(v))}{\text{eval}_{h,\sigma}(x.f) = h(v, f)} \quad (\text{R-Fld})$$

We use σ, a mapping from variables to values, to represent the stack frame of our program. For variable look up, we simply use $\sigma(x)$ to retrieve the value as long as $x \in \text{dom}(\sigma)$. For field update, we write in a style of $\sigma[x \rightarrow v]$, indicating mapping x to a new value v. Values are addresses or a special null address null_a. In the initialization of the stack, at the object creation or the method invocation, a boolean variable is also stored in the stack. "true" means the calling context is a constructor while "false" means the calling context is a instance method. This is used in helper function $\text{isInConstructor}(\sigma)$.

Heap h is a mapping from value to its type and fields, namely $v \rightarrow (T, \overline{f \rightarrow v'})$. We define a helper function $h_t(v)$ to get the type of the value v and $h(v, f)$ to get the value of filed f at address v. We use $\text{eval}_{h,\sigma}(e)$ to evaluate expressions. If the expression is a variable, we use already defined function $\sigma(x)$ to retrieve the value. If the expression is a field access, we first evaluate the receiver to a value and use helper function $h(v, f)$to get value of the field access.

R-New allocates a new space in the heap and stores object in it. The type of the object stored depends on the new operation's mutability qualifier which can only be `Mutable`, `Immutable`, `PolyMutable` and `RDM`, and that's how we break the new semantics rule into three cases. We use helper function $\text{conBody}(C)$ to retrieve the body statements of the constructor C, the evaluation is to assign values to the fields of class C. Note the evaluation of super() call is ignored but it's simple to add it. We just need to recursively find the conBody of the parent class and does evaluation there.

Because an object can only be mutable or immutable inside the heap, we break the field assignment to three sub cases: (1) R-FldAss mutates a mutable object (2) FldAss2 mutates an immutable object within the constructor or reassigning its field if the field is assignable(3) FldAssExc mutates an immutable object outside the constructor and gets an `ImmExc`.

R-Call relies on helper function MBody to get the body statements of the method. Once the new stack frame σ_1 is built up, we recursively do the program execution of the body statements. R-Cast permits upcast while R-CastExc disallows downcasts and incomparable casts. We do not allow downcasts like T-Downcast does, because in the runtime we can get the precise type of values.

To summarize, in this section, we first discuss about language features of PICO, then proceed to the formalization. The well-formedness rules and static type checking rules are consistent and synchronized with the type and validity checks in the PICO implementation. Last, we provide the operational semantics, modeling how the heap and stack correspondingly changes for each statement.

Chapter 4

Generics with Context-Sensitive Optional Type Systems

4.1 Exploring PICO-Generics

PICO inherits the generics behavior from Checker Framework, which enables PICO to typecheck generics code easily. For example,

```
1 class GenericFieldArray<E extends @Readonly Person> {
2     E[] elements;
3 
4     static void m(GenericFieldArray<@Immutable Person> r,
    GenericFieldArray<@Mutable Person> r2) {
5         // :: error: (illegal.field.write)
6         r.elements[0].f++;
7         r2.elements[0].f++;
8     }
9 }
```

On line 6, the type of `r.elements[0]` is resolved to `Immutable Person` and thus reassigning its field is not allowed. The post increment on line 7 is permitted as the type argument is mutable. This is good, showing that even without the mutability information at the declaration of the class (the field declaration is not annotated with any mutability qualifiers), the use of the field depends on the actual instantiation of the generic class. Things are still good if we write immutability qualifiers on type parameters upper bound, to strict the instantiation.

```
1  class MyMap<K,V extends @Immutable Object> {
2      K key; V val;
3
4      static void test() {
5          // :: error:(type.argument.type.incompatible)
6          MyMap<String, @Mutable MutableBox> mp = new MyMap<>();
7      }
8  }
```

Error is issued as expected on line 6 of the above code. However, we could break the type safety of PICO by writing like this:

```
1  class MyCollection<T> {
2      @Immutable T t;
3      MyCollection(@Immutable T p) {this.t = p;}
4      static void test() {
5          MyCollection<Box> mc = new MyCollection<>(new Box());
6          mc.t.f = 1;
7      }
8  }
9  /*@Mutable*/
10 class Box {int f;}
```

Listing 4.1: A use of a generic class breaks type safety

There is no error reported for the above code and it is hard to identify any dangers at the first glance. The generic class `MyCollection` is defined to have an immutable field `t`, but in actual invocation, a `Box` type is passed in, and suddenly the field is mutated at line 6. The caller may not know the immutability property of this generic class, and without errors reported from PICO and notice from the developers, the immutability contract is broken. The root cause is that by default `Box` is mutable and at invocation the `Mutable` qualifier overwrites all `Immutable` qualifier at the field declaration and method signature. We should fix this and also re-think when to use the declaration-site mutability information and when to use the use-site mutability information and we need to clarify the semantics of type qualifiers on these locations: type variable uses, type parameters and type arguments.

We realize this behavior is not PICO specific, and it is rooted in Checker Framework. In the following sections, we identify the problem, fix it and propose a model to formally reason the rules. We first show the methodology and then show the case study in immutability.

4.2 The General Problem and Solution

4.2.1 The General Problem

The aforementioned type safety issue inspires us to re-think about the semantics of optional types on type variables and how generics interact with optional type systems. Before introducing the formalization modeling the interaction of the two, we give an overview of the problem and solution.

```java
class MyCollection<@1 T extends @2 Object> {
    @3 T t;
    /*@4*/ T t2; // unannotated
    MyCollection(@5 T p) {this.t = p;}
    static void test() {
        MyCollection<@6 Box> mc = new MyCollection<>(new Box());
        mc.t.f = 1;
    }
}
```

Listing 4.2: Generics meets optional type systems

Listing 4.2 shows a generics class annotated with optional types, from `@1` to `@6`. Some questions would be

- What's the semantics of `@1` and `@2`? As these optional types are annotated on the type parameter, what's their relationship with optional types annotated on the type variable uses (`@3`)?
- If the type variable use is not explicitly annotated with a optional type (field `t2`), how should we handle this case?
- If the type provided along with the type argument (`@6`) is incompatible with the type specified on the type variable use (`@3`), which one should we adopt and which one should be discarded?
- In order to resolve the type of `mc.t`, we have to conduct two operations now. One is viewpoint adaptation, we adapt the type of `t` from the viewpoint of the receiver `mc`, and another is type variable substitution, we have to replace the type variables with corresponding type arguments. Which operation should be performed first?

Now we can see, optional types actually can be applied on much more places in the code and this makes the interaction of generics and optional type systems complicated. We need to give clear semantics to (1) annotated and unannotated type variable uses, type parameters; (2) how we do viewpoint adaptation and type variable substitution.

4.2.2 The Solution

First, we clarify the semantics of the optional types on each locations: (1) type parameters (`@1`, `@2` in listing 4.2); (2) type variable uses (`@3` and `@5`); (3) unannotated at all. We achieve this in part through adopting the new `@Sub` qualifier, which impacts the process of type variable substitution and viewpoint adaptation. `@Sub` stands for “substitute”, meaning that the type qualifier will be substituted by the type arguments’ qualifiers.

Annotated on type parameters `@1` and `@2` serve as the lower and upper bound of the type parameter `T`. That is, the optional type of the type argument (`@6`) used to substitute `T` should be the subtype of `@2` and the supertype of `@1`. This is the same as Checker Framework.

Annotated on type variable uses For example, `@3` and `@5` are annotated on the type variable uses, one is field declaration, and another is method parameter. This optional type is not restricted by the type parameter bounds, that means, `@3` could be the supertype of `@2` and the subtype of `@1`. More, if the type argument whose optional type is `@6` is passed in, the optional type of field `t` remains `@3`, meaning that we keep the declaration site optional type and discard the one provided along with the type argument. This is the same as Checker Framework, though implemented incorrectly in the context-sensitive part.

Unannotated at all For example, the field `t2` in the above listing. Now we apply the new optional type `@Sub` to be the default for these cases. For example, `@4` will be substituted by `@6`. Inside the class, the type variable use (`@4`) has the same upper and lower bounds as declared by the type parameter.

Second, we propose a model for viewpoint adaptation and type variable substitution. Essentially, we first categorize the viewpoint adaptation into three cases: (1) adapt a full type from a full type; (2) adapt a full type from an optional type; (3) adapt an optional type from an optional type. Then we first perform the viewpoint adaptation of case 2 and then do substitution. In the case of `mc.t`, we first do viewpoint adaptation adapting the type of `t` from the optional type of `mc`, then we do the substitution. The details of the process is described in section 4.3.2.

Inference Previously, type variables do not have a default type qualifier, and there is a need to distinguish between unannotated and annotated type variables, and find a way to encode this distinction. In order to achieve that, we need to use a special kind of constraint variable called `ExistentialVarSlot`. The semantics of the constraint variable is complicated and it makes difficulty to the constraint normalization process. After introducing `Sub`, every type variable has a default type which will largely simplify the constraint generation. This is a uniform treatment, because now inferring polymorphic code is the same as inferring monomorphic code.

To summarize, the contributions are as follows:

- A defaulting mechanism that will treat all unannotated generic type uses as annotated by `Sub` qualifiers.
- A viewpoint adaptation and substitution strategy to clarify the semantics of optional types on type variables. We believe the substitution is safe and satisfy developers' intent.
- An uniform inference strategy benefited from the treatment of type variables.
- A calculus along with formalization to verify the use of `Sub` qualifier. We further add `Lost` qualifier to the system, which means a type can not be expressed by the current type system.

4.3 Formalization

As we said earlier, we are trying to add a `Sub` qualifier (for better generics) and a `Lost` qualifier to Checker Framework. The syntax of our language are in Java style but simpler. For types, there would be monomorphic type and type variable. More, both of the two types can be annotated. The formalization aims at capturing and verifying the core part of Checker Framework generics.

4.3.1 Syntax

$$
\begin{aligned}
P &::= \overline{Cls}\ e \\
Cls &::= \text{class } C\langle \overline{X\ N}\rangle \text{ extends } C'\langle \overline{T'}\rangle\{\overline{T\ f}; \overline{mt}\} \mid \text{NullType} \mid \text{Object} \\
T, S &::= N \mid X \\
N &::= q\ C\langle \overline{T}\rangle \\
X &::= q\ \alpha \\
FTV &::= \alpha \text{ extends } N \text{ super } N' \\
U &::= \ C\langle \overline{T}\rangle \mid \alpha \\
q &::= \text{rdq} \mid \text{bot} \mid \text{top} \mid \text{sub} \mid \text{lost} \\
mt &::= \langle \overline{X\ N}\rangle T\ m(\text{this } N', \overline{x\ T'})\{s; \text{return } y\} \\
e &::= \text{null} \mid x \mid x.f \mid x.m\langle \overline{T}\rangle(y) \mid C.m\langle \overline{T}\rangle(y) \mid \text{new } N \mid (T)\ x \\
s &::= x.f = y \mid x = e \mid s; s' \\
\Gamma &::= \overline{\alpha \to \{q, N\}}; \text{this} \to N', \overline{x \to T}
\end{aligned}
$$

x, y, z	program variables
f	field identifier
m	method identifier
C	class identifier
NullType, Object	predefined class types
α, β	type variable identifier

A program `P` contains a set of classes and the main expression e as the entry of the program. `Cls` is a class declaration that is made of a class identifier `C`, a list of type parameters, an extends clause, a sequence of fields and methods.

Types are a bit complicated in this calculus. First we use `T, S` to range over all types. `T` is either a monomorphic type `N` or a type variable `X`. A monomorphic type `N` is composed of a primary qualifier `q`, a class identifier `C` and a set of type arguments `T`. A type variable `X` is made of a qualifier `q` and a type variable identifier α. `FTV` stands for full type of a type variable, which is only used in a helper function to represent the type ranges of type variable α. `U` stands for unannotated type, which is either an unannotated monomorphic type or an unannotated type variable. `q` represents one of the type qualifiers.

A method declaration mt contains a sequence of method type parameters, a return type, a method identifier, method parameters and method body. An expression e can be null, a variable, field access, instance method call, new operation and typecast. A statement s could be field assignment or variable assignment. We use x, y, z to range over a set of program variables. A program variable could be a this self reference, a method parameter or a local variable allocated on the stack. An environment Γ contains two kinds of mappings, one maps type variables to their qualifier lower bounds and NType upper bounds, another maps instance variables to their types. For any type parameter, the lower bound will be (q NullType), so, we omit the NullType in the mapping.

NullType is the bottom types of all monomorphic types, while Object is the super types of all monomorphic types.

4.3.2 Viewpoint Adaptation and Type Variable Substitution

We will discuss our viewpoint adaptation and substitution strategy in this subsection. To start with, we define a overloaded operator $\rhd$ that accepts two arguments. More specifically, $T \rhd\ T'$ will yield a new type by adapting T' from the viewpoint of T.

4.3.2.1 Qualifier w.r.t. Qualifier

The rules will be:

$$\begin{aligned} \rhd :: Q \times Q &\to Q \\ \top \rhd \mathrm{rdq} &= \mathrm{lost} \\ q \rhd \mathrm{rdq} &= q \qquad q \neq \top \\ q \rhd q' &= q' \qquad \text{otherwise} \end{aligned}$$

The first rule says rdq qualifier cannot depend on a $\top$ qualifier whose information is imprecise. We will also forbid the use of rdq in the static context by the well-formedness check. The second equation shows the use of rdq qualifier, which stands for “receiver dependent qualifier” and returns the receiver type if it is not the top type. In the rest scenarios, anything looks at q’ returns q’.

4.3.2.2 Qualifier w.r.t. Type

$$
\begin{aligned}
&\rhd :: Q \times \text{Type} \rightarrow \text{Type} \\
&\quad q \rhd X = (q \rhd q')\alpha && \text{where } X = q' \ \alpha && (4.3.1) \\
&\quad q \rhd N = (q \rhd q')C\langle\overline{q \rhd T}\rangle && \text{where } N = q' \ C\langle\overline{T}\rangle && (4.3.2)
\end{aligned}
$$

There are two conditions in this case, one is type variable, another is monomorphic type. In both cases, the adapted primary qualifier is decided by $q \rhd q'$, which we have discussed above. For the second case, we recursively apply the viewpoint adaptation to the type arguments.

In previous GUT paper, $\rhd_m$ is introduced to avoid unsoundness of the possible combinations of qualifiers, but that problem can be solved by the Lost qualifier.

4.3.2.3 Type w.r.t. Type

$$
\begin{aligned}
&\rhd :: \text{NType} \times \text{Type} \rightarrow \text{Type} \\
&\quad q \ C\langle\overline{S}\rangle \rhd T = \text{Substitute}((q \rhd T), \overline{S}, \overline{\beta}) && \text{where } \overline{\beta} = \text{dom}(C) && (4.3.3)
\end{aligned}
$$

$$
\begin{aligned}
&\text{Substitute}() :: \text{Type} \times \overline{\text{Type}} \times \overline{\text{TypeVar}} \rightarrow \text{Type} \\
&\qquad \text{Substitute}(\text{sub } \alpha, \overline{S}, \overline{\beta}) = \alpha[\overline{S/\beta}] && (4.3.4) \\
&\qquad \text{Substitute}(q \ \alpha, \overline{q' \ U}, \overline{\beta}) = q \ \alpha[\overline{U/\beta}] && (4.3.5) \\
&\qquad \text{Substitute}(q \ C'\langle\overline{T'}\rangle, \overline{S}, \overline{\beta}) = q \ C'\langle\overline{\text{Substitute}(T', \overline{S}, \overline{\beta})}\rangle && (4.3.6)
\end{aligned}
$$

This would be the most interesting case, which adapts a type from a viewpoint of an `NType` variable. We further define a helper function Substitute. This case is a bit different from the above two cases, as this incorporates the type variable substitution.

Substitute() accepts three arguments: (1) The first one is the viewpoint adaptation result. Note that this result has not performed the type variable substitution; (2) The second argument is a sequence of type arguments provided by the `NType` instance; (3) The third argument is a list of type parameters declared in the class C's declaration.

For formula(2), if the first argument is a type variable annotated with the sub qualifier, this annotated type variable will be replaced by corresponding annotated type, thus performing the type variable substitution; For formula(3), if the first argument is a type variable with qualifiers other than `Sub`, we keep the existing qualifier and substitute only the type variable with corresponding types; For formula(4), if the first argument is in `NType`, we keep the primary qualifier on that type (as the sub qualifier can only be applied on the type variable) and recursively perform substitution for the type arguments. All type arguments could be α-renamed to avoid variable capture.

4.3.3 Subtyping & Subclassing Rules

Before introducing the subtyping rules, we introduce some helper functions to get the actual full type of type variables.

$$\frac{\alpha \in \text{dom}(\Gamma) \qquad \Gamma \vdash \alpha : \{q', N\}}{\text{getFullType}(\Gamma, \text{sub } \alpha) = \alpha \text{ extends } N \text{ super } (q' \text{ NullType})} \quad (\textsc{GetFullType-1})$$

$$\frac{\alpha \in \text{dom}(\Gamma) \qquad \Gamma \vdash \alpha : \{q'', q' \; C'\langle\overline{T'}\rangle\} \qquad q \neq \text{sub}}{\text{getFullType}(\Gamma, q \; \alpha) = \alpha \text{ extends } (q \; C'\langle\overline{T'}\rangle) \text{ super } (q \text{ NullType})} \quad (\textsc{GetFullType-2})$$

Given an type environment Γ and an annotated type variable, GetFullType returns the type range of the type variable. The first one shows Sub α will have the same type as it is declared in the type parameter. The second function shows we will use the use-site qualifier `q` to override the information given by the declaration.

A concrete example is given below:

```java
class Demo<@L T extends @U Object> {
    @A T f1;
    @B T f2;
    /*@Sub*/ T f3;
    void test(Demo<@C Object> d) {
        d.f3;
    }
}
```

The type of `this.f1` should be `T extends @A Object super @A null`

The type of `this.f2` should be `T extends @B Object super @B null`

The type of `this.f3` should be `T extends @U Object super @L null`

The type of `d.f3` should be `@C Object`

Then we proceed to defining our subtyping and subclassing rules for our calculus.

$$\frac{\text{getFullType}(\Gamma, q\ \alpha) = \alpha \text{ extends } N \text{ super } N'}{(q\ \alpha)\ <:\ N} \quad (\text{ST-}\textsc{Extends})$$

$$\frac{\text{getFullType}(\Gamma, q\ \alpha) = \alpha \text{ extends } N \text{ super } N'}{N'\ <:\ (q\ \alpha)} \quad (\text{ST-}\textsc{Super})$$

$$\frac{}{\Gamma \vdash T <: T} \quad (\text{ST-}\textsc{Reflexivity}) \qquad \frac{T <: T' \qquad T' <: T''}{\Gamma \vdash T <: T''} \quad (\text{ST-}\textsc{Transitivity})$$

$$\frac{q <: q' \qquad C\langle\overline{T}\rangle \sqsubseteq C'\langle\overline{T'}\rangle}{\Gamma \vdash q\ C\langle\overline{T}\rangle <: q'\ C'\langle\overline{T'}\rangle} \quad (\text{ST-}\textsc{Annotated})$$

$$\frac{}{\Gamma \vdash \text{NullType} <:\ C\langle\overline{T}\rangle} \quad (\text{ST-}\textsc{Null})$$

As shown in rule ST-Extends, an annotated type variable is the subtype of the upper bound, declared by the `extends` clause. Similarly, the annotated type variable is the supertype of the lower bound, declared by the `super` clause. ST-Reflexivity and ST-Transitivity show our type system is reflexive and transitive. ST-Annotated declares the subtype relationships between annotated types. Last, according to ST-Null, NullType is the bottom of all unannotated monomorphic types.

The subclassing rule is defined as below:

$$\frac{\text{class } C\langle\overline{X\ N}\rangle \text{ extends } C'\langle\overline{T'}\rangle\{\overline{T\ f}; \overline{mt}\}}{C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{T'}\rangle} \quad (\text{SC-}\textsc{Extends})$$

$$\frac{}{C\langle\overline{T}\rangle \sqsubseteq C\langle\overline{T}\rangle} \quad (\text{SC-}\textsc{Reflexivity})$$

$$\frac{C\langle\overline{T}\rangle \sqsubseteq C''\langle\overline{T''}\rangle \qquad C''\langle\overline{T''}\rangle \sqsubseteq C'\langle\overline{T'}\rangle}{C\langle\overline{T}\rangle \sqsubseteq C'\langle\overline{T'}\rangle} \quad (\text{SC-}\textsc{Transitivity})$$

$$\frac{C\langle\overline{T}\rangle \sqsubseteq C'\langle\overline{T'}\rangle}{C\langle\overline{T}[T''/X'']\rangle \sqsubseteq C'\langle\overline{T'}[T''/X'']\rangle} \quad (\text{SC-}\textsc{Substitute})$$

$$\frac{T <: T' \qquad \text{lost} \notin T'}{T <:_s T'} \quad \text{(ST-Strict)}$$

$C\langle\overline{X}\rangle$ is the subclass of $C'\langle\overline{T'}\rangle$ if we have a class declaration `class C extends C’` (SC-Extends). SC-Reflexivity and SC-Transitivity show our subclassing system is reflexive and transitive. If two monomorphic types have subclass relationship, substituting their type parameters with same set of type arguments keep that subclass relationship (SC-Substitute). The strict subtyping rule, ST-Strict, is used to express the case that the supertype does not contain any Lost qualifiers.

4.3.4 Well-Formedness Rules

$$\frac{\begin{array}{c}\overline{\alpha \to \{q, N\}}; \emptyset \vdash \overline{N}, \overline{T},\ C'\langle\overline{T'}\rangle \text{ ok} \\ \overline{mt} \text{ ok in } C\langle\overline{X\ N}\rangle \qquad q \neq \text{sub}\end{array}}{\text{class } C\langle\overline{(q\ \alpha)\ N}\rangle \text{ extends } C'\langle\overline{T'}\rangle\{\overline{T\ f}; \overline{mt}\} \text{ ok}} \quad \text{(WFC)}$$

$$\frac{\begin{array}{c}\text{class } C\langle\overline{X\ N}\rangle \text{ extends } C'\langle\overline{T'}\rangle\{\overline{T\ f}; \overline{mt}\} \\ \Gamma \vdash \overline{T} \text{ ok} \qquad \Gamma \vdash \overline{T} <: \overline{(q\ C\langle\overline{T}\rangle \rhd N)} \\ \Gamma \vdash \overline{(q\ C\langle\overline{T}\rangle \rhd X)} <: \overline{T} \qquad q \neq \text{sub}\end{array}}{q\ C\langle\overline{T}\rangle \text{ ok}} \quad \text{(WFT-NVar)}$$

$$\frac{\begin{array}{c}\text{class } C\langle\overline{X\ N}\rangle \text{ extends } C'\langle\overline{T'}\rangle\{\overline{T\ f}; \overline{mt}\} \text{ ok} \\ \Gamma \vdash \overline{T} \text{ strictly ok} \qquad \Gamma \vdash \overline{T} <:_s \overline{(q\ C\langle\overline{T}\rangle \rhd N)} \\ \Gamma \vdash \overline{(q\ C\langle\overline{T}\rangle \rhd X)} <:_s \overline{T} \qquad q \neq \text{sub} \qquad \text{lost} \notin q\ C\langle\overline{T}\rangle\end{array}}{q\ C\langle\overline{T}\rangle \text{ strictly ok}} \quad \text{(SWFT-NVar)}$$

$$\frac{\alpha \in \text{dom}(\Gamma)}{\Gamma \vdash (q\ \alpha) \text{ ok}} \quad \text{(WFT-TVar)} \qquad \frac{\alpha \in \text{dom}(\Gamma) \qquad q \neq \text{lost}}{\Gamma \vdash (q\ \alpha) \text{ strictly ok}} \quad \text{(SWFT-TVar)}$$

$$\frac{\begin{array}{c}\Gamma = \overline{\alpha_m \to \{q_m, N_m\}}, \overline{\alpha_c \to \{q_c, N_c\}}; \text{this} \to (q\ C\langle\overline{sub\ \alpha}\rangle), \overline{x \to T_p} \\ \Gamma \vdash T_r, \overline{N_m}, \overline{T_p} \text{ ok} \qquad \Gamma \vdash y : T_r \qquad \text{override}(C, m) \qquad q_m, q_c \neq \text{sub}\end{array}}{\langle\overline{(q_m\ \alpha_m)\ N_m}\rangle T_r\ m(\text{this}\ (q\ C\langle\overline{\text{sub}\ \alpha}\rangle), \overline{x\ T_p})\{s; \text{return } y\} \text{ ok in }\ C\langle\overline{(q_c\ \alpha_c)\ N_c}\rangle} \quad \text{(WFM)}$$

$$\frac{\overline{Cls}\ \text{ok} \qquad \emptyset, \emptyset \vdash e : N}{\overline{Cls}\ e\ \text{ok}} \quad \text{(WFP)}$$

WFC defines the well-formedness of class declarations. The well-formedness should satisfy three conditions: (1) The upper bound of the class's type variable, the field types and the super class instantiation are well-formed; (2) methods are well-formed inside the class declaration; (3) `q` can not be `Sub`. That is, sub qualifier cannot be placed on type parameters.

For a monomorphic type $q\ C\langle\overline{T}\rangle$ to be well-formed, several requirements have to be met: (1) the type arguments of it, $\overline{T}$ have to be well-formed; (2) the viewpoint adapted upper bound, $q\ C\langle\overline{T}\rangle \rhd N$ is the supertype of the declared type argument $\overline{T}$; (3) the viewpoint adapted lower bound, $q\ C\langle\overline{T}\rangle \rhd X$ is the subtype of the declared type argument $\overline{T}$. Note here, we are using the receiver type to look at X, while the true lower bound should be q' NullType. However, changing the true lower bound to X does not affect the actual preconditions for this well-formedness rule and also make the rule simpler. Thus, we will use X in the viewpoint adaptation. (4) For an Ntype, sub qualifier is not allowed as the primary annotation. As for SWFT-NVar, it has stricter requirements: (1) demanding the field type to be strictly okay; (2) all subtype relationships should be strictly subtyped of the other; (3) `Lost` cannot appear in the instantiation of this NType.

According to rule WFT-TVar, as long as the type variable is in the domain of the type environment, any annotated type variable is well-formed. For SWFT-TVar, it further requires `q` to not be `Lost`.

The well-formed method rule, WFM, first builds up the type environment which contains two components, (1) mappings from type variables to their upper bounds; (2) mappings from variables to their types. Then it requires the return type, the upper bounds of the type parameters and the method parameters to be well-formed. More, method `m` respects the rule for overriding. For static methods, WFM-Static rule declares `rdq` should not appear in the method, including the type arguments, the upper bounds of the type arguments, the return type, the method parameters and the method expression. we can see all type arguments of `this` have `Sub` qualifiers. This is useful when doing substitution, as Sub $\rhd q = q$.

Then if all classes are well-formed, the program is considered as well-formed (WFP).

4.3.5 Static Type Checking

Below are rules for type checking the context-sensitive generic language.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad (\text{T-Var}) \qquad \frac{\Gamma \vdash e : T \qquad T <: \Gamma(x)}{\Gamma \vdash x = e : T} \quad (\text{T-VarAss})$$

$$\frac{N \neq \top \; C\langle\overline{T}\rangle}{\Gamma \vdash \text{new } N : N} \quad (\text{T-New}) \qquad \frac{}{\Gamma \vdash \text{null} : \text{NullType}} \quad (\text{T-Null})$$

$$\frac{\Gamma \vdash T \text{ ok}}{\Gamma \vdash (T)\ e : T} \quad (\text{T-Cast}) \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash e : T'} \quad (\text{T-Subs})$$

$$\frac{\Gamma \vdash x : N \qquad \text{FType}(N, f) = T}{\Gamma \vdash x.f : T} \quad (\text{T-Read})$$

$$\frac{\Gamma \vdash x : N \qquad \text{FType}(N, f) = T \qquad \Gamma \vdash y :_s T}{\Gamma \vdash x.f = y : T} \quad (\text{T-Write})$$

$$\frac{\begin{array}{c}\Gamma \vdash z : N_r \qquad \Gamma \vdash \overline{T_l} \text{ strictly ok} \\ \text{MSig}(N_r, m, T_l) = \langle\overline{X\ N}\rangle T_r\ m(\overline{x\ T_p}) \\ \Gamma \vdash \overline{y} :_s \overline{T_p} \qquad \overline{T_l} <:_s \overline{N} \qquad \overline{X} <: \overline{T_l}\end{array}}{\Gamma \vdash z.m\langle\overline{T_l}\rangle(\overline{y}) : T_r} \quad (\text{T-Call})$$

$$\frac{\Gamma \vdash e : T \qquad \text{lost} \notin T}{\Gamma \vdash e :_s T} \quad (\text{T-Exp-Strict})$$

$$\frac{\Gamma \vdash x : T \qquad \text{lost} \notin T}{\Gamma \vdash x :_s T} \quad (\text{T-Var-Strict})$$

$$\frac{\text{FType}(\text{ClassOf}(N), f) = T' \qquad N \rhd T' = T}{\text{FType}(N, f) = T} \quad (\text{FType})$$

$$\frac{\text{class C } \langle\_\rangle \text{ extends } \_ \ \{\_ \text{ T f } \_;\ \_\ \}}{\text{FType}(N, f) = T} \quad (\text{FType})$$

$$\dfrac{\begin{array}{c}\overline{N_r \rhd N' = N} \qquad \overline{N_r \rhd X' = X} \\ N_r \rhd T'_r = T_r \qquad \overline{N_r \rhd T'_p = T_p} \\ \text{MSig}(\text{ClassOf}(N_r), m) = \langle \overline{X'\ N'} \rangle T'_r\ m(\overline{x\ T'_p})\end{array}}{\text{MSig}(N_r, m, \overline{T_l}) = \langle \overline{X\ N} \rangle T_r\ m(\overline{x\ T_p})} \quad (\text{MSig})$$

$$\dfrac{\text{class C } \langle \_ \rangle \text{ extends } \_ \ \{\_;\ \_\ \text{ms}\{\text{e}\}\ \_\ \} \qquad \text{MName(ms)} = \text{m}}{\text{MSig}(C, m) = \text{ms}} \quad (\text{MSig})$$

Rule T-Var looks up in the type environment Γ for the type of the variable. T-VarAss requires the subtype relationship between the left-hand-side and the right-hand-side. This variable assignment rule shows that the values of expressions can be stored to variables. T-Cast allows the typecast operations as long as the target type is well-formed. The subsumption rule, T-Subs allows to apply subtyping to `e`'s type implicitly. For T-Read, we first look up the receiver type and then use helper function FType to get the adapted field type as the result. FType is defined to first look up the declared type of that field and use the receiver type to get the viewpoint adaptation result. Note that FType uses overloading, the second FType retrieves the declared field type from the class declaration. `ClassOf(N)` yields the class of the monomorphic type N. Rule T-Write permits a field assignment as long as the right hand side can be strictly typed to be T.

Rule T-Call is a bit sophisticated. It gets the adapted method signature by using MSig and requires the method arguments to be strictly typed as $\overline{T_p}$, type arguments to be the subtype of the upper bound and the supertype of the lower bound. Helper function MSig is also overloaded. The first one gets the non-adapted method definition and uses the receiver type to adapt the type parameter bounds (both lower bound and upper bound), the return type as well as the method parameters. The second MSig asks for the class name and method name and returns the declared method signature. The invocation of static methods is described by rule T-Call-Static, which is very similar to T-Call.

Last, same as before, we need strict typing rules to type expressions and variables.

4.3.6 Operational Semantics

We use σ, a mapping from variables to the values, to represent the stack frame of our program. We write $\sigma(x)$ for the variable lookup (if $x \in dom(\sigma)$) and $\sigma[x \to v]$ for stack update (meaning x is assigned with a value v). Values are addresses or a special null address $null_a$.

Similarly, we define heap $h = \overline{v \to o}$ as a mapping from addresses to objects. An object $o = (T, \overline{f \to v})$ contains its type and a mapping from field identifiers to the values stored in the fields. We use $h(v.f)$ to access the value of an object's field, given the object's address and the field name and use $h[v.f = v']$ to update a particular field to a new value. $h(v)$ is defined to return the object, its type and the mapping.

We demonstrate our big-step operational semantics using judgements $e, \sigma, h \rightsquigarrow v, \sigma', h'$, indicating an expression e is evaluated under stack σ and heap h, resulting a value v, modified stack σ' and heap h'

$$\frac{\sigma(x) = v}{x, \sigma, h \rightsquigarrow v, \sigma, h} \quad \text{(R-Var)} \qquad \frac{e, \sigma, h \rightsquigarrow v, \sigma, h'}{x = e, \sigma, h \rightsquigarrow v, \sigma[x \to v], h'} \quad \text{(R-VarAss)}$$

$$\frac{}{null, \sigma, h \rightsquigarrow null_a, \sigma, h} \quad \text{(R-Null)}$$

$$\frac{x, \sigma, h \rightsquigarrow v, \sigma, h \qquad h(v) = (T, ...)}{(T)\ x, \sigma, h \rightsquigarrow v, \sigma, h} \quad \text{(R-Cast)}$$

$$\frac{\begin{array}{c} v \notin dom(h) \qquad \text{ClassOf(N)} = \text{C} \\ \text{fields(C)} = \overline{f} \qquad h' = h[v \to (N, \overline{f \to null_a})] \end{array}}{new\ N, \sigma, h \rightsquigarrow v, \sigma, h'} \quad \text{(R-New)}$$

$$\frac{x, \sigma, h \rightsquigarrow v, \sigma, h \qquad h(v.f) = v'}{x.f, \sigma, h \rightsquigarrow v', \sigma, h} \quad \text{(R-Read)}$$

$$\frac{y, \sigma, h \rightsquigarrow v, \sigma, h \qquad x, \sigma, h \rightsquigarrow v_0, \sigma, h \qquad h' = h[v_0.f = v]}{x.f = y, \sigma, h \rightsquigarrow v, \sigma, h'} \quad \text{(R-Write)}$$

$$\frac{\begin{array}{c} z, \sigma, h, \rightsquigarrow v, \sigma, h \qquad \overline{y}, \sigma, h, \rightsquigarrow \overline{v_1}, \sigma, h \qquad \text{cls}(z) = C \qquad \text{MBody}(C, m) = e \\ \text{MSig}(C, m) = \langle\_\rangle T_r\ m(\overline{x\ T_p}) \qquad e, \sigma[\overline{x} \to (v : \overline{v_1})], h \rightsquigarrow v_2, \sigma', h' \end{array}}{z.m\langle T\rangle(\overline{y}), \sigma, h \rightsquigarrow v_2, \sigma, h'} \quad \text{(R-Invk)}$$

For the evaluation of a program variable `x` we simply look up the stack (R-VAR). For variable assignment, after evaluating the expression, we update the variable in the stack with the new value (R-VARASS). Expression *null* evaluates to $null_a$ (R-NULL). For R-CAST, we get the value of the expression and retrieve its type. If the type is the same as the target type, the casting is allowed.

For object creation R-NEW, we first ensure that address v does not exist in h before. Then we allocate an object with type N and fields with $null_a$ addresses.

For R-READ and R-WRITE, one looks up the heap for the value, another one updates a particular key-value pair in the heap.

In R-INVK we use $cls(z)$ to retrieve the class name of the variable and then use `MBody`, `MSig` to get the method's body and signature. Note that parameter list $\overline{x}$ implicitly contains `this` variable, and we use $(v : v_1)$ to concatenate two sets of values. So, $\sigma[\overline{x} \to (v : \overline{v_1})]$ means assigning `this` with value v and assigning the rest method parameters with $\overline{v_1}$.

4.4 Case Study in Immutability

PICO is a context-sensitive type system which supports Java generics. It occurs to us naturally to do case study in this system to test our general model proposed earlier.

Let's re-examine listing 4.1 and check if this time PICO can correctly issue a `type.incompatible` error. On line 5, there is a constructor call and the original constructor signature is generic, so it needs to be substituted. The receiver of the call has type `MyCollection<@Mutable Box>` and the method parameter waits to be adapted and substituted is `@Immutable T`. This is adapting a type variable from the viewpoint of a monomorphic type, and by applying formula 4.3.5 in section 4.3.2.3 we have $\text{Substitute}(\text{@Immutable } T, \text{@Mutable Box}, T) = \text{@Immutable } T[\text{Box}/T] = \text{@Immutable Box}$. After substitution, the constructor signature is `MyCollection(@Immutable Box p)` while the argument has type `@Mutable Box`. Thus a `type.incompatible` error is issued as expected. The same logic applies to line 6, where `mc.t` will be resolved to type `@Immutable Box` and forbid the mutation.

4.4.1 Annotated Type Variable Uses

The use of `Supplier` functional interface is worthwhile to explore. Assume we have a piece of code below, where `Demo` is a parameterized class with type parameter `T` upper bounded by `Readonly RDMBox`. It declares a field with type `Supplier<@Mutable> T` and has corresponding initialization and getter method. Last, we instantiate the `Demo` class inside the static method.

```
1  class Demo<E extends @Readonly RDMBox> {
2      // Supplier has one abstract method, whose signature is "E get();"
3      Supplier<@Mutable E> f;
4  
5      Demo(Supplier<@Mutable E> p) {
6          this.f = p;
7      }
8  
9      @Mutable E createContent() {
10         return f.get();
11     }
12 
13     static void test() {
14         Demo<@Immutable RDMBox> e = new Demo<>(@Immutable RDMBox::new);
15     }
16 }
17 
18 @ReceiverDependentMutable
19 class RDMBox{
20     int v = 0;
21 }
```

Let's first look at the constructor body. The receiver is `this` parameter and has type `@Mutable Demo<E extends @Readonly RDMBox>` and the field's type is declared on line 3. Previously, an `assignment.type.incompatible` error is thrown, because the viewpoint adaptation in PICO uses the `E extends @Readonly` information to incorrectly overwrite the type argument of `@Mutable E` in `f`. The detailed unexpected error message is: " `found Supplier<E extends @Mutable ...> but required Supplier<E extends @Readonly ...>`"

The improved viewpoint adaptation now type checks the statement on line 6. The computation steps are as follows:

$$\begin{aligned}
&\text{@Mutable Demo}\langle E \text{ extends @Readonly RDMBox}\rangle \rhd \text{@Mutable Supplier}\langle\text{@Mutable } E\rangle \\
&= \mathit{Substitute}(\text{@Mutable} \rhd \text{@Mutable Supplier}\langle\text{@Mutable } E\rangle, \text{@Sub } E, E) && (4.4.1)\\
&= \mathit{Substitute}(\text{@Mutable Supplier}\langle\text{@Mutable } E\rangle, \text{@Sub } E, E) && (4.4.2)\\
&= \text{@Mutable Supplier}\langle\mathit{Substitute}(\text{@Mutable } E, \text{@Sub } E, E)\rangle && (4.4.3)\\
&= \text{@Mutable Supplier}\langle\text{@Mutable E}[E/E]\rangle && (4.4.4)\\
&= \text{@Mutable Supplier}\langle\text{@Mutable } E\rangle && (4.4.5)
\end{aligned}$$

The first step is derived by using formula 4.3.4; we apply formula 4.3.2 in the second step; at the third step, we use rule 4.3.6; for the forth and fifth steps we apply *Substitute* function recursively and get the final adapted type. Note that the second argument of *Substitute* in the second step has type `Sub` according to our WFM rule.

Now the statement in the constructor body type checks. We further examine the object creation statement in the static method. The constructor signature is adapted from the perspective of `Demo<@Immutable RDMBox>`, the declared type of the object. The constructor signature stays the same after the viewpoint adaptation process. The method reference passed in is `@Immutable RDMBox <init>()`, while the required type is `@Mutable RDMBox get(@Mutable Supplier<@RDMBox> this)`, and thus typecheck fails as expected(return type incompatible). Previously PICO unsoundly permits this statement.

4.4.2 Inference of Unannotated Type Variable Uses

Here we have a piece of code which does not have type variables explicitly annotated.

```java
class MyCollection<T extends Box> {
    T t;
    void fun() {
        this.t.mutate();
    }
}
@Mutable
class Box {}
```

Listing 4.3: Inference for unannotated generics code

Previously, `t` will not have a default type qualifier, and we use a special constraint variable `ExistentialVariableSlot` to encode the distinctions of unannotated and annotated type variables, which can lead to complicated and incorrect constraints encoding. Let's dig deeper into this example to see how constraints are generated. The above code is first processed to the below intermediate form.

```
1 class InferenceT<@VarAnnot(1) T extends @VarAnnot(2) Box> {
2     @VarAnnot(3) T f;
3     void fun() {
4         this.t.mutate();
5     }
6 }
7 @Mutable
8 class Box {}
```

In order to invoke the `mutate` operation successfully at line 4, `this.t` should be the subtype of `@Mutable`. However, the constraint is not directly applying to `@VarAnnot(3)` but a special kind of constraint variable `@4`, which is not explicitly written in the above code. This special constraint variable is the `ExistentialVariableSlot` and represents the following things (note I ignore the viewpoint adaptation for simplicity):

```
Constraint variable @4:
  if ( 3 exists ) {
    3 <: @Mutable
  } else {
    2 <: @Mutable
  }
```

As the constraint is based on this special constraint variable, we need a constraint normalization approach to encode such constraints, which makes the process even harder. More, we should consider cases that `@2` could also not exist, then `@4` will be even complicated containing nested if-else clauses.

By introducing the `Sub` qualifier, every type variable is annotated with type qualifier. Now we just need to use `VarSlot` qualifier to encode it (which is the same as monomorphic types), and thus a uniform treatment and can simplify constraint generation. For the above code, we just need a simple constraint `@3 <: @Mutable` to encode the method invocation on line 4.

4.4.3 Simple Constraint Generation Rules

Based on the `Sub` qualifier and the viewpoint adaptation rules given, we devise simple constraint generation rules without using the `ExistentialVariableSlot`. The rules are the inference counterpart of the type checking rules, focusing on the generics aspect and enforcing mutability properties. We use Σ to denote the set of constraints, and other symbols are consistent with the syntax in the type checking part. Our constraint-based approach is inspired by GUT-Infer[9].

$$\frac{\begin{array}{c} q\ C\langle\overline{T}\rangle \text{ ok} \qquad \text{class } C\langle\overline{X\ N}\rangle \cdots \text{ ok} \\ \Sigma_1 = \{\overline{(q\ C\langle\overline{T}\rangle \rhd X)} <: \overline{T}\} \qquad \Sigma_2 = \{\overline{T} <: \overline{(q\ C\langle\overline{T}\rangle \rhd N)}\} \end{array}}{\Gamma \vdash q\ C\langle\overline{T}\rangle : \Sigma_1 \cup \Sigma_2} \quad (\text{Infer-NVar})$$

$$\frac{\Sigma_{preference} = \{q = \text{Sub}\}}{\Gamma \vdash (q\ \alpha) : \Sigma_{preference}} \quad (\text{Infer-TVar})$$

$$\frac{\Gamma(x) = T \qquad \Sigma = \{T' <: T\}}{\Gamma \vdash x = e : T' : \Sigma} \quad (\text{Infer-VarAss})$$

$$\frac{\begin{array}{c} \Gamma(x) = \{q, N\} \qquad \text{FType}(N, f) = T \\ \text{initialized}(x) \qquad \Sigma_1 = \{T' <: T\} \qquad \Sigma_2 = \{N <: \text{Mutable}\} \end{array}}{\Gamma \vdash x.f = e : T' : \Sigma_1 \cup \Sigma_2} \quad (\text{Infer-Write})$$

$$\frac{\Gamma \vdash e : T' \qquad \Sigma = \{T' <: T\}}{\Gamma \vdash (T)\ e : \Sigma} \quad (\text{Infer-Cast})$$

Infer-NVar shows that an instantiation of a generic class (the instantiation could be a field or method parameter) adds constraints to the type parameter upper and lower bounds. Infer-TVar adds a preference constraint to `q`, giving `q` a higher chance to be `Sub`, and will be adapted according to instantiations. Infer-Write adds a mutability and initialization constraint to `x`.

By employing these rules, we can encode constraints for type variables using just `VarSlot` and this also shows PICO, a context-sensitive optional type system, now has an uniform treatment of type variables in type inference.

To conclude this section, we first start out with a type safety issue in PICO and then look at the general problem, which is how should optional type systems interact with generics. We point out the problems and solutions and carry out the formalization. Last, we come back to the immutability type system, verifying our new proposed model that does work for this context-sensitive type system.

Chapter 5

Conclusion and Future Work

In this thesis, we design a context-sensitive optional type system which supports generics well. We achieve this by introducing a new model of viewpoint adaptation and type variable substitution which has clear semantics. Based on that, we fix the limitation in the Checker Framework. We also present a language and corresponding formalization to verify our approach (including well-formedness rules, type rules) and we show this works for our context-sensitive immutability type system. We also devise a new set of rules to encode constraints on generics code in PICO, which is simpler than the old approach.

For the immutability type system, Practical Immutability for Classes and Objects (PICO), we give an overview and identify some problems in it and fix them. The enhancements either improve the flexibility of the tool or solve type safety issues in the old system. We then formalize the core part of PICO, providing type rules and operational semantics of it. We also give a review of immutability in the background section.

There are more design choices in the interaction between generics and optional type systems. For example, the decision of keeping the optional type of the type argument or the type variable use, and we have provided our solution in section 4.2.2. One interesting design choice would be distinguishing between weaker and stronger types in this scenario and let the stronger type override the weaker type. In the type lattice, a more precise type is usually considered as a stronger type.

Continuing working on the formalization of PICO would be an interesting future direction. Even though we provide the operational semantics, modeling the heap and stack update, but interesting immutability properties are left unproved. The proof is usually done by proving the progress and preservation theorems. For progress theorem, it's relatively easy, as we just need to make sure program process successfully without running into any immutability exception as shown by rule R-FldAssExc. It's harder to prove the preservation theorem, as we need to show that our static types are still preserved after a step of reduction in the runtime semantics. Mathematically, if $\Gamma \vdash h, \sigma$ and $h, \sigma, s \rightsquigarrow h', \sigma'$, then we still have $\Gamma \vdash h', \sigma'$. This is hard because statically we have these language constructs: polymorphism (the PolyMutable qualifier), context sensitivity (RDM qualifier), reference mutability (Readonly qualifier), assignability and initialization types, while dynamically heap only stores immutable and mutable objects. It would be challenging but worthwhile to bridge the dynamic and static part. Moreover, we can use coq to provide a mechanized and formal proof to go beyond the pen and paper proof.

We present type rules and well-formedness rules of PICO but haven't provided a complete set of inference rules for the inference counterpart. In the future, we could formalize the constraint generation providing the implementation with a formal specification.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive Software Verification-the KeY Book. *Lecture notes in computer science*, 10001, 2016.

[2] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. NullAway: Practical Type-Based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 740–750, New York, NY, USA, 2019. Association for Computing Machinery.

[3] James Gosling; Bill Joy; Guy Steele; Gilad Bracha; Alex Buckley; Daniel Smith; Gavin Bierman. *The Java Language Specification Java SE 17 Edition*. Oracle, 2021.

[4] Gilad Bracha. Pluggable Type Systems. OOPSLA 2004 Workshop on the Revival of Dynamic Languages, 2004.

[5] 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, 2017.

[6] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring Language Support for Immutability. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 736–747, 2016.

[7] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanc Muslu, and Todd W. Schiller. Building and Using Pluggable Type-Checkers. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 681–690, 2011.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, page 28–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming*, pages 333–357. Springer, 2011.

[10] EISOP. Checker Framework Manual. https://eisop.github.io/cf/manual/manual.html. Accessed: 2023-10-09.

[11] Google. Error Prone. https://errorprone.info/index. Accessed: 2023-10-09.

[12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 21–40, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Guava: Google Core Libraries for Java. https://github.com/google/guava. Accessed:2023-12-08.

[14] Wei Huang, Yao Dong, and Ana Milanova. Type-Based Taint Analysis for Java Web Applications. In *Fundamental Approaches to Software Engineering: 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 17*, pages 140–154. Springer, 2014.

[15] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 879–896, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, may 2001.

[17] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and Precision by Combining Expressive Type Systems and Deductive Verification. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[18] Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. Jml reference manual, 2008.

[19] ShuChen Liu. A Lightweight Type System with Uniqueness and Typestates for the Java Cryptography API, 2023.

[20] Ana Milanova, Wei Huang, and Yao Dong. CFL-Reachability and Context-Sensitive Integrity Types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, page 99–109, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[22] David J. Pearce. Jpure: A modular purity system for java. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2011.

[23] Jamie Quinonez. *Javarifier: Inference of reference immutability in Java.* PhD thesis, Massachusetts Institute of Technology, 2008.

[24] Dimitri Racordon and Didier Buchs. A Practical Type System for Safe Aliasing. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, page 133–146, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Immutability we can afford. https://elizarov.medium.com/immutability-we-can-afford-10c0dcb8351d. Accessed: 2023-10-09.

[26] Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. CiFi: Versatile Analysis of Class and Field Immutability. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 979–990, 2021.

[27] Tobias Runge, Marco Servetto, Alex Potanin, and Ina Schaefer. Immutability and Encapsulation for Sound OO Information Flow Control. *ACM Trans. Program. Lang. Syst.*, 45(1), mar 2023.

[28] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[29] 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.

[30] Lian Sun. An Immutability Type System for Classes and Objects: Improvements, Experiments, and Comparisons, 2021.

[31] Mier Ta. Context Sensitive Typechecking and Inference: Ownership and Immutability, 2018.

[32] The EISOP JDK. https://github.com/eisop/jdk. Accessed: 2023-12-08.

[33] 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.

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

[35] Weitian Xing, Yuanhui Cheng, and Werner Dietl. Ensuring correct cryptographic algorithm and provider usage at compile time. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP '21, page 43–50, New York, NY, USA, 2021. Association for Computing Machinery.

[36] ZK SpreadSheet. https://github.com/zkoss/zkspreadsheet. Accessed:2023-12-08.