Context Sensitive Typechecking And Inference: Ownership And Immutability

by

Mier Ta

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, 2018

© Mier Ta 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

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

Abstract

Context sensitivity is one important feature of type systems that helps creating concise type rules and getting accurate types without being too conservative. In a context-sensitive type system, declared types can be resolved to different types according to invocation contexts, such as receiver and assignment contexts. Receiver-context sensitivity is also called viewpoint adaptation, meaning adapting declared types from the viewpoint of receivers. In receiver-context sensitivity, resolution of declared types only depends on receivers' types. In contrast, in assignment-context sensitivity, declared types are resolved based on context types to which declared types are assigned to.

The Checker Framework is a poweful framework for developing pluggable type systems for Java. However, it lacks the ability of supporting receiver- and assignment-context sensitivity, which makes the development of such type systems hard. The Checker Framework Inference is a framework based on the Checker Framework to infer and insert pluggable types for unannotated programs to reduce the overhead of manually doing so. This thesis presents work that adds the two context sensitivity features into the two frameworks and how those features are reused in typechecking and inference and shared between two different type systems — Generic Universe Type System (GUT) and Practical Immutability for Classes And Objects (PICO).

GUT is an existing light-weight object ownership type system that is receiver-context sensitive. It structures the heap hierarchically to control aliasing and access between objects. GUTInfer is the corresponding inference system to infer GUT types for unannotated programs. GUT is the first type system that introduces the concept of viewpoint adaptation, which inspired us to raise the receiver-context sensitivity feature to the framework level. We adapt the old GUT and GUTInfer implementation to use the new framework-level receiver-context sensitivity feature. We also improve implicits rules of GUT to better handle corner cases.

Immutability is a way to control mutation and avoid unintended side-effects. Object immutability specifies restrictions on objects, such that immutable objects' states can not be changed. It provides many benefits such as safe sharing of objects between threads without the need of synchronization, compile- and run-time optimizations, and easier reasoning about the software behaviour etc. PICO is a novel object and class immutability type system developed using the Checker Framework with the new framework-level context sensitivity features. It transitively guarentees the immutability of the objects that constitute the abstraction of the root object. It supports circular initialization of immutable objects and mutability restrictions on classes that influence all instances of that class. PICO supports creation of objects whose mutability is independent from receivers, which inspired us to add the assignment-context sensitivity feature to the framework level. PICOInfer is the inference system that infers and propagates mutability types to unannotated programs according to PICO's type rules.

We experiment PICO, PICOInfer and GUTInfer on 16 real-world projects up to 71,000 lines of code in total. Our experiments indicate that the new framework-level context sensitivity features work correctly in PICO and GUT. PICO is expressive and flexible enough to be used in real-world programs. Improvements to GUT are also correct.

Acknowledgements

I'd like to first thank my supervisor Professor Werner Dietl. This thesis would be not possible without his enormous help and guidance. I also want to say thank you to my thesis readers Professor Gregor Richards and Professor Arie Gurfinkel for their time to read this thesis and all the feedbacks and comments to make this thesis better. I'm thankful to all my colleges in our research group especially Zhuo Chen, Jeff Luo, and Jianchu Li for their help and support during the two years of my study. Last but not the least, I thank all my friends who brought joy and happiness to me and made me a better person.

Dedication

This is dedicated to my parents who always support me no matter what happens.

Table of Contents

List of Figures ix

1 Introduction 1

1.1 Motivation . . . 3

1.2 Contribution . . . 4

1.3 Thesis Organization . . . 4

2 Background And Related Works 5

2.1 Background on Checker Framework . . . 5

2.2 Background on Checker Framework Inference . . . 7

2.3 Background on Generic Universe Type System (GUT) . . . 9

2.4 Background on Freedom Before Commitment (FBC) Type System . . . 12

2.5 Background on Immutability . . . 14

2.6 Related Work . . . 16

2.6.1 Javari . . . 16

2.6.2 ReIm And ReImInfer . . . 18

2.6.3 Google Error Prone Immutable . . . 20

3 Context Sensitivity 22

3.1 Introduction . . . 22

3.2 Receiver-context Sensitivity . . . 23

3.3 Assignment-context sensitivity . . . 24

3.4 Interaction Between Receiver-Context Sensitivity And Assignment-Context Sensitivity 27

3.5 Implementation Details . . . 27

3.5.1 Receiver-context sensitivity . . . 27

3.5.2 Assignment-context sensitivity . . . 28

4 Practical Immutability For Classes And Objects (PICO) Type System 29
4.1 Overview . . . 29
4.2 Qualifiers And Hierarchies . . . 30
4.2.1 Mutability Hierarchy . . . 31
4.2.2 Assignability Dimension . . . 32
4.3 Viewpoint Adaptation Rule . . . 32
4.4 Motivating Examples . . . 33
4.4.1 Immutable Object And Its Creation . . . 33
4.4.2 Receiver-Context Sensitivity . . . 33
4.4.3 Assignment-Context sensitivity . . . 34
4.4.4 Separation of Assignability and Mutability . . . 34
4.4.5 Transitive Immutability Guarantee . . . 34
4.4.6 Exclude Fields From Abstract State . . . 35
4.4.7 Initialization of Immutable Objects . . . 36
4.5 Abstract State . . . 37
4.6 Implicits . . . 39
4.7 Defaults . . . 39
4.7.1 Initialization Defaults . . . 40
4.7.2 Mutability Defaults . . . 40
4.7.3 Assignability Defaults . . . 42
4.8 Language Design . . . 42
4.8.1 Valid New Instance Creation Types . . . 42
4.8.2 Type Element Bound . . . 42
4.8.3 Compatability With Super Type Element . . . 43
4.8.4 Fields . . . 44
4.8.5 Field Initializations . . . 44
4.8.6 Constructors . . . 45
4.8.7 Cirular Initialization of Immutable Objects . . . 47
4.8.8 Compatiblity Between Constructor And Type Element Bound . . . 48
4.8.9 Compatability Between Current Constructor and This/Super Constructor . . . 48
4.8.10 Compatability Between Type Usage With Type Element Bound . . . 50
4.8.11 Instance Methods . . . 50
4.8.12 Instance Methods Invocations . . . 51
4.8.13 Instance Methods Overriding . . . 51

4.8.14 PolyMutable Methods And Their Resolutions . . . 53
4.8.15 Static Context . . . 55
4.8.16 Possible Loophole Of Assignable Fields . . . 55
4.8.17 Arrays . . . 56
4.8.18 Type Casts . . . 57
4.9 Formalization . . . 57
4.9.1 Language Syntax Definition . . . 58
4.9.2 Type Environment . . . 58
4.9.3 Subtype Relations . . . 59
4.9.4 Helper Function . . . 59
4.9.5 Viewpoint Adaptation Rules . . . 59
4.9.6 Typing Rules . . . 59
4.9.7 Well-formedness Rules . . . 60
4.9.8 Extension to Real Java With Static And Blocks . . . 61

5 Implementation And Experiments — PICO 63
5.1 Implementation . . . 63
5.1.1 PICO Type Checker . . . 63
5.1.2 PICOInfer . . . 64
5.2 Experiments . . . 65
5.2.1 PICO Type Checker . . . 67
5.2.2 PICOInfer . . . 71

6 Improvements to the Generic Universe Type System 85
6.1 Implementation Improvements . . . 85
6.2 Implicit Bottom Types . . . 86
6.3 Viewpoint Adaptation To Bottom Receiver Problems . . . 86
6.4 Experiments - GUTInfer . . . 87
6.4.1 Benchmarks . . . 87
6.4.2 Inference Results . . . 88
6.4.3 Checker Framework Inference Statistics . . . 89
6.4.4 Solver and Timing Statistics . . . 89

7 Problems And Future Work 91

8 Conclusions 93

References 94

List of Figures

2.1 GUT qualifier hierarchy . . . 10
2.2 Topology graph of the `Person` class example . . . 10
2.3 Topology graph of the compound expression example . . . 11
2.4 Viewpoint adaptation rules for GUT . . . 12
2.5 Qualifier hierarchy of FBC . . . 13
2.6 Type rules for field update in form x.f = y . . . 13

4.1 Qualifier hierarchy of mutability . . . 31
4.2 Field declarations and abstract state . . . 37
4.3 Syntax of language . . . 58
4.4 Revised and extended syntax of language . . . 61

5.1 Benchmark projects for running PICO and PICOInfer . . . 65
5.2 Categorization of errors by PICO type checker on unannotated benchmarks . . . 67
5.3 Number of @UnderInitialization and @Assignable added . . . 75
5.4 Number of each qualifier in the best combination of settings . . . 75
5.5 Number of each qualifier without manual @UnderInitialization annotations inserted 79
5.6 Inferred results with @UnderInitialization . . . 79
5.7 Inferred results without @UnderInitialization . . . 79
5.8 Number of each qualifier with optimistic assumption for unchecked methods . . . 81
5.9 Number of each qualifier without PreferenceConstraints . . . 81
5.10 Numbers of Slots and Constraints (and PreferenceConstraints) generated . . . 82
5.11 Solver related statistics and timing information with and without PreferenceConstraint 84

6.1 Numbers of `peer` and `rep` qualifiers inferred for static topology and owner-as-modifier 88
6.2 Numbers of other ownership qualifiers inferred for static topology and owner-as-modifier . . . 88
6.3 Numbers of Slots and Constraints generated . . . 89
6.4 Solver statistic and timing information for static topology . . . 90

Chapter 1

Introduction

In software programs, depending on the specific choice of programming language, variables, expressions, functions, etc. are assigned some properties, which are called types, that specify some requirements or expectations we have on those language constructs. For example, in Java, we can declare:

```
int x = 0;
```

By this, we not only allocate a variable called `x` in the stack and assign value 0 to `x`, we also restrict `x` to only hold integer values. If by accident or due to programmer's lack of knowledge of Java, a string value is assigned to `x`, then the Java compiler would report the error, saying the assignment is not compatible, and doesn't allow the program to compile and run. Otherwise, if other language constructs depended on the assumption that `x` is guaranteed to only hold integer values and performed numeric calculations over `x`, then the calculation would show undefined behaviors on strings. During the compilation phase, the Java compiler detects this ill-typed program and won't compile it. This is the result of Java's built-in type system.

In general, the purpose of type systems is to restrict programs to a subset for which the behavior is defined and predictable. Java's type system already helps filtering out programs that are unable to execute.

However, there can be even smaller subsets where more properties are guaranteed. For example, a subset from which programs never throw NullPointerException that stops the program immediately. This can be achieved by additional type checking on dereferences of `null`. One such existing type system is the Nullness Type System[11]. For example, a programmer can write:

```
@NonNull Object o = new Object();
o.toString();
```

Then after the "extended type", @NonNull Object, is enforced semantically, `o.toString()` is guaranteed to not throw NullPointerException, because `o` is guaranteed to store a non-`null` value (a reference to an object on the heap). In a malformed program, if `null` is assigned to `o`, then the compiler would report the error and stop compiling the buggy program, just like in the previous example, a string is not allowed to be assigned to `x` which should only hold integer values.

The Nullness Type System is an example of pluggable type systems for Java. Pluggable type systems for Java allow extending standard Java types and running additional type checkers that check the type soundness of the extended types, to enforce additional properties in many domains. This additional checking is optional, meaning that programmers can choose whether to enable it or not. Besides, pluggable type systems are orthogonal, so they don't depend on each other[1]. Programmers can choose whatever number of additional type checks they want to conduct to get even stronger guarantees on their programs. For example, a programmer can choose to check dereference-of-null, regular expression validity and concurrency-safety one after one to make sure no such bugs exist in their code. Using pluggable type checking, programmers can get more confidence on their code.

There are many applications of pluggable type systems to different domains. In addition to the type systems we mentioned before, the applications also include detecting SQL injections, forbidding unsafe operating system commands to be executed, ensuring correct usage of units to perform meaningful arithmetic operations, and so on.

The Checker Framework[3] is the framework we use in this thesis for pluggable type checking. It is a framework to facilitate the development of pluggable type systems for Java. Type system developers can declare their own type qualifiers using Java's annotation syntax since JavaSE 8, and implement their own type checkers to enforce type rules of their type systems.

If the pluggable type checking checks whether existing pluggable types obey the type rules or not, the inference is the process of giving a set of well-typed pluggable types to programs, such that humans don't need to manually insert them. The Checker Framework Inference[4] is a framework that is based on the Checker Framework to infer annotations (pluggable types's concrete implementation) to unannotated or partially-annotated programs, to reduce the overhead of manual annotation effort.

This thesis focuses on two applications of pluggable type checking and inference: ownership and immutability. They are implemented as Generic Universe Type System (GUT)[25] and Practical Immutability For Classes And Objects Type System (PICO), based on the Checker Framework and Checker Framework Inference.

Ownership[16] is a way to structure the object store and to control aliasing and modifications of objects[25]. Normally, relations among objects on the heap are complex: an object might reference another object, which again might reference other objects transitively. As a result, there might be a huge, complex network over objects on the heap. But since the network is not intentionally organized, we can't gain too much interesting knowledge from them or enforce additional rules based on the topology of those objects. Ownership is one of the ways to gain more control over the object network, specifically to control aliases and side-effects: it creates a topology by which objects "own" other objects and mutation is restricted based on this ownership. Generic Universe Type System is an existing light-weight ownership type system. In GUT, we can enforce a principle called the owner-as-modifier, such that only the owner objects can directly mutate the owned objects. The main benefit of this is to prevent aliases to the objects from mutating the objects as aliases are not the owners so unintended side-effects can be effectively captured.

Immutability[16] is proposed by many experts to be a good software practice. It is a way to control mutations to objects, to avoid unintended side-effects. Immutable objects are objects

[1]Type systems don't depend on each other. However, a type system may be composed of several sub type systems.

whose internal states don't change in their lifetime[7]. Usually, their states are set during construction phase and will be "freezed" after constructions finish[2]. There are many benefits to immutable objects. In concurrent programming, immutable objects are very useful because their state cannot change, so they can't be corrupted by thread interference or observed in inconsistent states. Immutable objects also make reasoning about the software easier. It's widely accepted that maximum reliance on immutable objects is a sound strategy to create simple and reliable code. We developed a novel object immutability type system, PICO, which provides transitive immutability guarantees over the abstraction of objects that are interesting to clients of the objects. It is based on an existing initialization-tracking type system called Freedom Before Commitment (FBC)[23], to handle initializations of objects correctly.

Context sensitivity is an important feature for type systems. In a context sensitive type system, types of accessed elements aren't always the declared types. Instead, their types are determined by both "context types" and declared types. In this thesis, we show two kinds of context sensitivity: receiver-context sensitivity and assignment-context sensitivity. Viewpoint adaptation is first introduced by GUT, to get declared types from the "viewpoint" of receivers. We generalize viewpoint adaptation to the high level concept, receiver-context sensitivity. In receiver-context sensitivity, the context is receivers of the accessed elements such as fields, methods etc. In assignment-context sensitivity, the context is the target to which the invoked methods are assigned. They both help creating concise and polymorphic type systems and are core concepts for the two type systems, GUT and PICO, in the thesis.

1.1 Motivation

1. The logic of performing viewpoint adaptation was tightly coupled to GUT, thus can't be shared by other type systems that are also receiver-context sensitive. Decoupling this logic and raising it to the Checker Framework will make the development of such type systems much easier.

2. We'd like to let the Checker Framework Inference support viewpoint adaptation too. Reusing the viewpoint adaptation logic from the Checker Framework in Checker Framework Inference will further reduce code duplications.

3. Polymorphic qualifiers are qualifiers that get resolved based on actual arguments to method invocations in the Checker Framework[3]. However, the current resolution of polymorphic qualifiers in the Checker Framework doesn't consider the assignment context, therefore they are not sensitive to assignment contexts. We want to add assignment-context sensitivity feature to the resolution of polymorphic qualifiers to the framework, which can be easily inherited by type systems that need it.

4. We want to explore the possibility of creating a new practical immutability type system using the two new/updated framework-level context-sensitivity features, as a proof of the correctness of them. The type system should be easy to understand and use, and expressive enough to reflect the real-world need.

[2] Note that constructions don't necessarily mean constructors. Constructions may continue after the constructors finish.

[3] They can only be used on method declarations

5. The implementation of GUT is outdated and doesn't work on the latest frameworks anymore. Especially, the latest Checker Framework Inference introduces a new solver framework called Type Constraint Solver[21]. But the inference for GUT, GUTInfer, wasn't adapted to this change. We want to update GUT and GUTInfer to make them workable again. This also helps proving whether the framework-level receiver-context sensitivity feature is implemented correctly.

6. In GUT, boxed primitive types and String types are not handled consistently compared to primitive types: arbitrary ownership modifiers can be used on boxed primitive and String types, but they just get ignored in subtype relations. This thesis aims to make ownership types of those three consistent.

1.2 Contribution

1. We implemented receiver-context sensitivity and assignment-context sensitivity features in the Checker Framework, and adapted the receiver-context sensitivity implementation to generate constraints in inference mode in Checker Framework Inference. In the future, type systems that need either of or both forms of context sensitivity can inherit the logic from the framework without any cost.

2. We created, implemented, formalized and experimented a novel immutability type system called PICO, which provides transitive immutability guarantees over abstractions of objects, instead of every in-memory details of fields. Its expressiveness and flexibility make it stand out from other existing immutability type systems.

3. We updated the GUT implementation to the latest frameworks, and removed code duplications by combining the typechecking and inference implementations into one class for each component. We also adapted GUTInfer to the Type Constraint Solver. GUT and GUTInfer became workable again, and can help to make sure that the changes to Checker Framework and Checker Framework Inference are correct by showing none of GUT and GUTInfer breaks.

1.3 Thesis Organization

Chapter 2 discusses background knowledge that is needed to read the thesis. They include: Checker Framework, Checker Framework Inference, Generic Universe Type Systems, Freedom Before Commitment type system, immutability, and several existing immutability type systems' implementations. Chapter 3 introduces what receiver-context sensitivity and assignment-context sensitivity are in detail. Chapter 4 explains theoretical aspects of PICO type system in detail. Chapter 5 shows implementations and experiments of PICO and PICOInfer on real-world projects. Chapter 6 presents improvements to GUT type system and experiments of GUTInfer on real-world projects. Chapter 7 lists problems and future work. Finally, Chapter 8 summarizes all the work in this thesis.

Chapter 2

Background And Related Works

This chapter explains the background and related work that are needed to read the thesis or inspired us. Section 2.1 and section 2.2 introduce Checker Framework and Checker Framework Inference, which are two frameworks for implementing pluggable type systems for Java. Section 2.3 and section 2.4 talks about two existing type systems: GUT for ownership and FBC for initialization, respectively. Section 2.5 discusses what immutability is and classifies immutability systematically. Section 2.6 presents related work about immutability that inspired us, using three existing immutability type systems/tools.

2.1 Background on Checker Framework

Checker Framework[3] is a framework to facilitate developing pluggable type systems that enhances Java's type system to make it more powerful and useful. Type system developers can easily implement their type systems without worrying about underlying low level jobs, such as parsing the source code and traversing the Abstract Syntax Tree (AST). Users can select any checker distributed with Checker Framework or third party checker to detect and prevent bugs, such as the dereference of `null` bug that throws NullPointerException.

Since JavaSE 8 release, annotations can be used on type uses rather than only on declarations[13]. This change made it possible for us to define custom annotations, attach semantic meanings for them and use them in the source code to extend Java's type system. Checker Framework's idea is to use annotations as the extended types to standard Java types. Checker Framework also has modules to parse Java source code to AST, and provides a standard implementation to traverse the AST and perform the additional typechecking and basic validity checking. If any AST node violates the type or well-formedness rules of a particular type system, Checker Framework reports the error on the accurate position of the line that caused this violation.

For example, if we have source code:

```
class A {
  void foo() {
    Object o = null;
    o.toString();
  }
}
```

In this example, we want to detect dereference-of-`null` bug, so we will run Nullness Checker[11]. As a pluggable type system, Nullness Checker is passed as a command line flag to javac:

```
javac -processor nullness testinput/A.java
```

The running result is:

```
testinput/A.java:4: error: [dereference.of.nullable] dereference
    of possibly
-null reference o
        o.toString();
        ^
1 error
```

To create a new type system, type system developers need to do the following[6]:

- Define qualifiers/annotations and the qualifier hierarchy: a qualifier represents the extended type that the type system needs to support. Among different qualifiers, there should also be a hierarchy between them. This hierarchy tells which qualifier is the subtype of which qualifier, and what the top qualifier and bottom qualifier are.

- Type rules: the standard subtyping rule is already enforced in the framework level, so developers don't need to have special handling for subtype checks. This will be automatically done once the qualifier hierarchy is provided. Developers only need to specify what type-system-specific type rules to enforce. For example, when dereferencing a field in the Nullness Type System, the receiver should be @NonNull.

- Type introduction rules: specify which qualifiers are treated as if being present if expressions don't have explicit qualifiers. For example, for literals except `null` literal, for example, String.class, they are implicitly @NonNull. The default qualifier for unannotated references is @NonNull.

- Dataflow rules: Checker Framework uses Dataflow Framework to perform flow-sensitive refinement to be more accurate when doing typechecking. For example, even though local variables can be declared with the top qualifier, if there is an assignment to that local variable, then that local variable will be treated as if it is declared with the rhs qualifier when being read. If the type system has special dataflow rules, then type system developers can override the default rules.

- Interface to the compiler: specifies what qualifiers are supported, what options are supported by the type system, etc.

From the implementation point of view, a type system in Checker Framework typically is composed of the below major components:

- `Checker` class: the main entry point to plug into the standard javac compiler. It creates all the other dependent classes that are needed to perform the typechecking. It also lists what checker options it supports.

- `AnnotatedTypeFactory`: a factory that queries qualifiers with standard Java types, AnnotatedTypeMirror, from trees and elements. Explicit qualifiers, implicits, defaults, and dataflow refinement are applied in order.
- `Visitor`: an class that is implemented with visitor pattern to traverse the AST and enforce type rules. Additional type-system-specific rules are enforced in this class.
- `Validator`: ensure types are well-formed. It checks the validity of types to make sure invalid types won't be passed.
- `Analysis`, `Value`, `Transfer`: dataflow related classes.

2.2 Background on Checker Framework Inference

Checker Framework Inference[4] is a framework on which inference for a type system can be developed. Checker Framework checks if the existing qualifiers violate type rules or not. Even if there is no explicit qualifier on some type uses, they are still implicitly applied or defaulted to a certain qualifier, according to the type introduction rule discussed before. We call this typechecking. However, Checker Framework Inference is to infer the non-existing qualifiers so that the inferred result still satisfies the type rule of the type system. By this, we could reduce the overhead for type system users to manually annotate the source code, which is a big problem for legacy code or projects of large size.

The idea of inference is to generate constraints while traversing the AST of source code, instead of checking whether existing types are consistent with type rules. Since there are no existing qualifiers in code, inference uses a place holder to represent the qualifier on the type-use locations. Then depending on the kind of AST node, constraints between those place holders are generated, Constraints represent the restriction among solutions, which is to make the solutions still obey the type rules. After all the ASTs are visited, and constraints are generated, constraints will be serialized so that solver can convert it to the domain-specific problem that it's designed for. For example, MaxSAT solver converts the constraints into a CNF form as a boolean satisfaction problem and gives the solutions by finding a set of truth values for all variables in the CNF. After the solver solves the constraints, as the final step, the solver's result will be translated back to qualifiers understandable by Checker Framework.

We use u_i to represent a slot, the placeholder for source code locations that need to infer qualifiers. i is the ID that uniquely determines each slot. Currently, Checker Framework Inference supports these constraints:

1) `SubtypeConstraint`($u_1 <: u_2$): the solution qualifier for u_1 should be a subtype of the solution qualifier for u_2.

2) `EqualityConstraint`($u_1 = u_2$): the solutions for u_1 and u_2 should be the same.

3) `InequalityConstraint`($u_1 \neq u_2$): the solutions for u_1 and u_2 should be different.

4) `ComparableConstraint`($u_1 <:> u_2$): the solutions for u_1 and u_2 should be comparable. Comparable means $u_1 <: u_2$ or $u_2 <: u_1$.

5) `CombineConstraint`($u_3 = u_1 \triangleright u_2$): the solutions for u_3 should be the result of viewpoint adapting the declared type u_2 to the receiver type u_1. What viewpoint adaptation is and how it works will be discussed in detail in section 3.2.

6) `PreferenceConstraint`($u \sim= c$): make the solution for u1 be the same as c, if possible. c represents ConstantSlot, a special slot whose value is already known.

7) `ImplicationConstraint`($c_1 c_2 ... c_{n\text{-}1} \Rightarrow c_n$): If constraints c_1, c_2 ... $c_{n\text{-}1}$ all hold, c_n should also hold. $c_1 c_2 ... c_{n\text{-}1} c_n$ are arbitrary constraints from any of the above constraints except `PreferenceConstraint`. This is the only composite constraint currently supported.

As an example, we illustrate an inference process using Dataflow Type System [21]. Dataflow Type System can perform data-flow analysis by inferring all possible run-time Java types of method return types, method parameters, fields, and variables at compile time. If we have completely unannotated source code, and we would like to infer Dataflow qualifiers:

```
1 Object foo() {
    if (...) {
3     return new Object();
    } else {
5     return 3;
    }
7 }
```

After running the inference for Dataflow Type System, one slot will be placed on the method return type. No Slot is inserted to new instance creation statement on line 3 or literal return statement on line 5, as the type rules of Dataflow type system implicitly treat them to have type @Dataflow(typeNames="Object") on line 3 and @Dataflow(typeNames="Integer") line 5. The slot insertion result is as below:

```
1 @1 Object foo() {
    if (...) {
3     return new Object();
    } else {
5     return 3;
    }
7 }
```

The generated constraints are:

```
1 @Dataflow(typeNames="Object") <: @1
  @Dataflow(typeNames="Integer") <: @1
```

According to the type rules of Dataflow Type System, slot @1 will be inferred to solution @Dataflow(typeNames="Object", "Integer"), which is the supertype of both @Dataflow(typeNames="Object") and @Dataflow(typeNames="Integer"). Annotation-tools, a tool we use to insert inferred solutions back to the source code, then inserts the solution to the return type of foo() method. After the insertion, the code snippet will be annotated with typecheckable qualifiers from Dataflow type system:

```
  @Dataflow(typeNames="Object", "Integer") Object foo() {
2   if (...) {
      return new Object();
4   } else {
```

```
    return 3;
  }
}
```

Right now, Checker Framework Inference supports four different kinds of solvers: Sat4J[12], Lingling[9], LogicBlox[10] and Microsoft Z3[18]. They use different theories to encode constraints to equivalent solver domain-specific problems. Users can specify which solver to use, as for a particular type system, there may be a backend which is the most suitable.

In this thesis, we only use Sat4J solver. Sat4j solver is a Java library for solving boolean satisfaction and optimization problems. It can solve SAT, MAXSAT, Pseudo-Boolean, Minimally Unsatisfiable Subset (MUS) problems.

2.3 Background on Generic Universe Type System (GUT)

Generic Universe Type System[25] is a lightweight object ownership type system that describes and enforces heap topology in a hierarchical structure. Optionally it supports the owner-as-modifier principle, so that only an owner object can change the state of objects that belong to it.

Generic Universe Type System defines the below qualifiers to represent static ownership information of the referred-to object relative to the current receiver `this`[19]:

1) `peer`: referred-to object by `peer` reference has the same owner as current receiver object `this`.

2) `rep`: referred-to object by `rep` reference belongs to the current object `this`. Or, we say the current receiver `this` owns the object referred by this `rep` reference. One object has at most one owner object. Objects that have the same owner are within the same context. Ownership is not transitive — if a owns b, b owns c, a doesn't own c.

3) `any`: doesn't provide static information about the relationship of the referred-to object relative to the current receiver `this`. This is the top qualifier of GUT.

4) `lost`: expresses that a relationship exits, but is not able to be expressed by `peer` or `rep`. The difference between `lost` and `any` is that `lost` represents a certain static ownership information, but it cannot be expressed by `peer` or `rep`; however, `any` express no static ownership information at all.

5) `self`: used only for type of current receiver object `this`.

6) `bottom` (not shown in the hierarchy): bottom qualifier of GUT. Used for primitive types that are not owned by anyone. Subtype of `self` and `rep`.

The qualifier hierarchy for GUT is shown in figure 2.1:

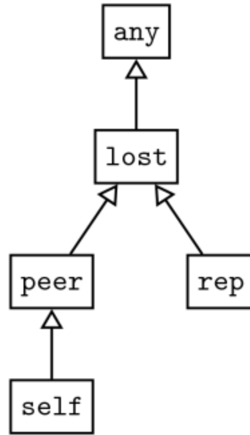

Figure 2.1: GUT qualifier hierarchy

One example from *Tunable Static Inference for Generic Universe Types*[19] is:

```
1 public class Person {
    peer Person spouse;
3   rep Account savings;
    rep List<peer Person> friends;
5   int assets() {
      any Account a = spouse.savings;
7     return savings.balance + a.balance;
    }
9 }
```

The topology graph of the `Person` class example is illustrated in figure 2.2:

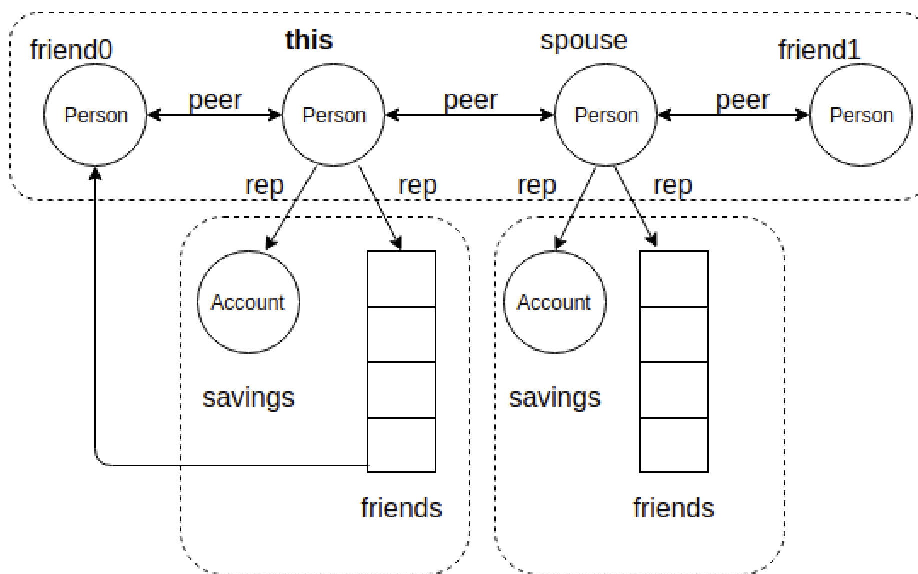

Figure 2.2: Topology graph of the `Person` class example

spouse object is a peer of the current receiver object this: savings belongs to the current receiver object this; friends is a list that belongs to current receiver object this, while elements stored inside are Person objects that are peer to current receiver object this.

Compound expressions' types are evaluated by combining ownership qualifiers of its components. For example:

```
class A {
  rep Person tony;
  void foo() {
    ... tony.spouse;// Field access compound expression
  }
}
```

Topology graph of the compound expression example is shown in figure 2.3:

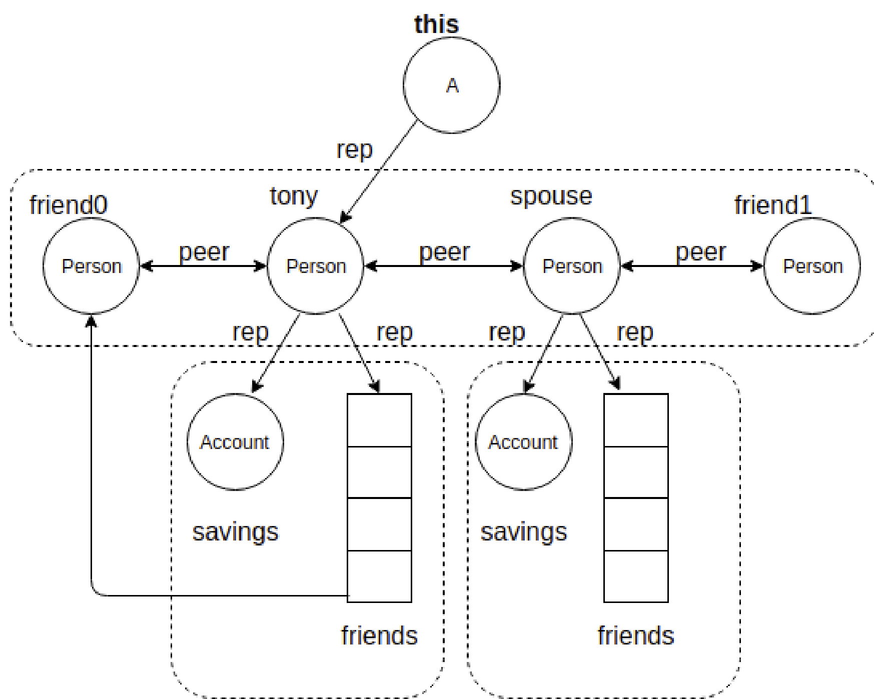

Figure 2.3: Topology graph of the compound expression example

tony is rep to the current receiver this, spouse is peer to tony, thus tony.spouse also has the same owner as tony (they are in the same context). Therefore, tony.spouse has ownership type rep. This operation is called viewpoint adaptation — it adapts the declared type of spouse to type of tony, which is the result of getting the type of spouse from the "viewpoint" of tony to the current receiver this.

Viewpoint adaptation rules define what every combination of receiver type and declared type yields as result types. Figure 2.4 shows the viewpoint adaptation rules for GUT:

Receiver \ Declared	any	lost	peer	rep	self	bottom
any	any	lost	lost	lost	lost	bottom
lost	any	lost	lost	lost	lost	bottom
peer	any	lost	peer	lost	lost	bottom
rep	any	lost	rep	lost	lost	bottom
self	any	lost	peer	rep	self	bottom
bottom	any	lost	lost	lost	lost	bottom

Figure 2.4: Viewpoint adaptation rules for GUT

Because there are so many combinations that yield `lost`, different `losts` may not be the same, it doesn't make sense to write to a `lost` type. GUT forbids `lost` on left-hand-side of assignments, for example, variables (fields and local variables) and method parameters (pseudo-assignments).

The owner-as-modifier principle forbids mutation (re-assigning field, calling side-effecting methods etc.) through `lost` or `any` receiver. Only the owner object (`rep`) or objects in the same context (`peer`) can initiate mutation. This avoids aliases to the target object that are not owners from side-effectingly modifying the object's state.

For example, A `Person` object can mutate `savings.balance` because `savings` is an `Account` object owned by `Person` object and so the owner has the permission to mutate `savings` object. This is reflected in implementation as `savings` having `rep` type. However, if there is an alias to the `Account` object, e.g. `any Account alias`, then assignment to `alias.balance` is forbidden, because `alias` is `any` and no ownership information is known so this assignment is not allowed. By the owner-as-modifier principle, it's enough that only owners guarantee invariants to hold on representation (`rep`) objects, as there won't be other places to mutate those representation objects (every non-owner references will get type `lost` according to the figure 2.4).

2.4 Background on Freedom Before Commitment (FBC) Type System

Freedom Before Commitment Type System[23] is a type system that statically tracks the initialization state of objects, so that unsafe dependency on invariants based on the assumption that the receiver object is fully initialized is forbidden when the receiver object is under initialization.

It's very important to track initialization of objects for some type systems. Take the Nullness type system as an example: even if a field of a class is @NonNull, until the receiver object of that class finishes initializing all the fields, there is always a time frame in which some/all fields haven't been initialized and the references stored inside the fields are still `null`. As a result, if the constructed receiver object is passed to a method or stored in the field of another object, which assumes all the fields of the under-initialization object are @NonNull, and dereferences any of them, then there is still possibility to throw NullPointerException. This is because the invariant that field is @NonNull hasn't been established yet, until the receiver object of that @NonNull field is fully initialized.

Freedom Before Commitment Type System is to address this problem by tracking each object's state as either `committed` or `free`. `Committed` means that the object is already fully initialized

and invariants based on the full initialization of that object can be safely established, but `free` means the object is still being initialized. `unclassified` is the top qualifier of FBC. Figure 2.5 shows the qualifier hierarchy of FBC:

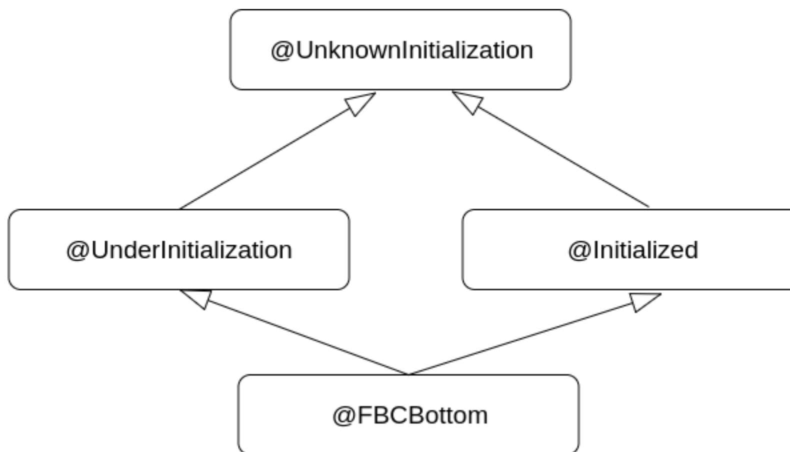

Figure 2.5: Qualifier hierarchy of FBC

The meaning of each qualifier is:

- @UnknownInitialization: concrete implementation of `unclassified`. Initialization state is unknown.
- @Initialized: concrete implementation of `committed`. Objects of this type are deeply initialized.
- @UnderInitialization: concrete implementation of `free`. Objects of this type are in the process of initialization. The invariant hasn't been established yet. For example, @NonNull fields of @UnderInitialization objects may still be null.
- @FBCBottom: bottom qualifier that makes the type hierarchy complete. Used on corner cases such as `null` literals.

FBC type system has a transitive/deep initialization guarantee: if one object is `committed`, then all the fields are also `committed`. That means that all transitively reachable objects from the current `committed` object are also `committed`. This transitive guarantee needs to have well-designed type rules, so that objects in correct initialization state are stored in `committed` and `free` objects respectively, to not break the deep initialization guarantee. Figure 2.6 shows which types of initialization states of objects can be stored safely to fields of receivers with different initialization states:

x \ y	committed	free	unclassified
committed	✓	x	x
free	✓	✓	✓
unclassified	✓	x	x

Figure 2.6: Type rules for field update in form x.f = y

`this` in constructors are implicitly `free`, because the constructor's responsibility is initializing objects. FBC also argues that if all the constructor parameters are of type `committed`, then the created object is `committed`, otherwise it's `free`. The proof of this can be found in *Freedom Before Commitment: A Lightweight Type System for Object Initialisation* paper[23].

In general, FBC type system's annotation overhead is low. Nearly all types are `committed` in programs and the default type is `committed`, which greatly reduces the amount of explicitly written `committed`. @UnderInitialization, the concrete implementation of `free`, is only needed in very few cases, such as when constructors call instance methods, for example, init(), then the declared receiver of that instance method needs to be @UnderInitialization. Methods declared with @UnderInitialization shouldn't assume the receiver object is already initialized, otherwise, an error will be reported.

2.5 Background on Immutability

Immutability is a way of controlling mutations to avoid unintended side effects. There are many existing immutability type systems. They have different definitions of immutability, and different levels of immutability and different goals. Before continuing the topic of immutability, we need to first systematically define 7 dimensions of immutability according to the classification introduced by *Exploring Language Support for Immutability* [17] and the Immutability chapter of *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*[16].

1. Reference immutability vs Object immutability: reference immutability applies mutability restrictions to references to objects, rather than objects themselves. A mutable reference can be used to modify the referred-to object, such as re-assigning its fields or calling side-effecting methods; a read-only reference is forbidden to be used to mutate the referred-to object. By reference immutability, one can avoid mutating states using a reference. However, an object referenced by a read-only reference might still be mutated by other mutable alias references. Object immutability applies to the object itself. All references to an immutable object are not allowed to modify the state of the immutable object. Object immutability subsumes reference immutability, as references are also checked against their immutability contracts and no violations are allowed.

2. Class-based immutability vs Object-based immutability: class-based immutability applies to every instance of a class. All instances of an immutable class are immutable. Object-based immutability applies to a particular instance of the class. In this case, a class might have both mutable instances and immutable instances.

3. Non-transitive immutability vs Transitive immutability: non-transitive/shallow immutability only guarantees direct fields of an immutable object are not re-assignable. However, objects stored in field may be mutated. However, transitive/deep immutability provides a stronger guarantee, such that object collections that are reachable by following references from the fields of an immutable object are also immutable. One example from *Aliasing in Object-Oriented Programming: types, analysis, and verification*[16]:

```
class C {
  D f;
```

```
}
class D {
  int x;
}

immutable C c;
c.f = otherD;// error in both deep and shallow immutability
c.f.x++;// error in deep immutability; allowed in shallow
    immutability
```

4. Immediate immutability vs Delayed immutability: in immediate immutability, immutability guarantee holds as soon as the constructor finishes. Delayed immutability allows initialization of an immutable data structure to continue after constructors finish.

5. Abstract immutability vs Concrete immutability: abstract immutability permits benevolent side-effects that don't affect the abstraction of the objects, such as caches, debug options, rearrangement of elements, etc. Not all the details of the fields are visible to the clients. One example is an internal cache field that saves time by avoiding duplicate, expensive re-computations. In abstract immutability, the abstract state is protected by the immutability guarantees. It refers to the set of objects that are part of the abstraction of the root object. The cache field above is outside the abstract state. However, concrete immutability requires that every in-memory field can not be changed. Therefore, even if the cache field's state changes, the object enclosing the cache is also considered changed in concrete immutability.

6. Static enforcement vs Dynamic enforcement: static enforcement is only at compile-time, while dynamic enforcement requires a run-time component to actually guarantee the type soundness at run-time, using the most-concrete types of objects. There are advantages and disadvantages of both. The advantage of static enforcement is that it doesn't introduce any runtime overhead. However, since lacking run-time checks, some language features such as downcasts, and covariant array component subtypes may not be safe at run-time, but the static type system can do nothing about them. Dynamic enforcement makes sure at run-time, program behavior obeys the immutability guarantees, but having additional run-time checks slows down the performance.

7. Polymorphic vs non-polymorphic: in a polymorphic implementation, there can be a single method signature that operates on multiple mutability types. Parametric polymorphism is one relevant example. It allows using polymorphic mutability types on parameters so that the method can accept inputs with different mutability types. However, in a non-polymorphic proposal, declared types of those parameters should at least be the least upper bound of all possible arguments for method parameters (similarly for fields).

One important thing to note is the difference between assignability and mutation. Assignability is a property of variable: it specifies if a variable is allowed to be re-assigned or not. Assignment to a variable doesn't mutate the original object referred to. One example from *Aliasing in Object-Oriented Programming: types, analysis, and verification* [16],

```
Date myVar = ....;// local variable
...
```

```
myVar = anotherDate;// Reassignment doesn't mutate old Date
   object
```

Reassignment to a field isn't mutation to the object referred by the field, but instead, is mutation to the object that contains the field. For example,

```
myClass.itsDate = anotherDate; // myClass object is mutated
```

Java forbids re-assignment of `final` fields, but doesn't forbid mutation to objects referred by `final` fields, such as re-assigning fields of the `final` fields or calling side-effecting methods on the `final` fields.

2.6 Related Work

In this section, we present several existing immutability type systems/tools that inspired us to develop PICO. They don't interact with each other, nor are they directly used by PICO.

2.6.1 Javari

Javari[24] is a static type system that expresses and enforces reference immutability constraints. It transitively guarantees that the abstract state of objects referred by readonly references cannot be changed.

Consider the following code snippet from `JDK1.1.1`:

```
class Class {
  private Object[] signers;
  Object[] getSigners() {
    return signers;
  }
}
```

`signers` is a private field, but the getSigners() method returns the internal `signers` array directly to callers, so malicious callers might tamper with the array by adding arbitrary objects into the `signers` array.

However, if we specify the mutability of the returned array as `readonly` according to the syntax of Javari:

```
class Class {
  private Object[] signers;
  Object readonly [] getSigners() {
    return signers;
  }
}
```

Then the callers can only have readonly access to the returned array, thus any mutation to the internal array is not allowed. Javari classifies a reference as either mutable or readonly. Javari is a type system that enforces reference immutability and provides abstract immutability, supporting excluding fields from abstract states by declaring them as assignable or mutable.

Javari clearly decouples mutability and assignability of fields. For mutability, it defines qualifiers:

1) mutable: a reference that can be used to mutate the referred-to object. Mutable fields are excluded from the abstract state.

2) readonly: a reference that can't be used to modify the abstract state of the referred-to object. However, non-abstract state fields are still allowed to be mutated via readonly reference.

3) this-mutable: a polymorphic reference whose mutability restriction inherits from the receiver object. Only allowed on instance fields. this-mutable fields of mutable reference are mutable; this-mutable fields of readonly reference are readonly, thus mutations are not allowed.

For assignability, Javari defines:

1) assignable: a reference is always re-assignable. If used on a field, that field is excluded from the abstract state, no matter what the mutability type of the receiver object is.

2) final: Java's keyword to restrict a reference to be assigned only once. final fields must be initialized in constructors as Java requires.

3) this-assignable: a reference's assignability inherits from the receiver object. this-assignable fields of mutable objects are resolved to assignable; of readonly objects are equivalent to final.

Separating mutability and assignability increases orthogonality of them and makes Javari more expressive. For example, previously, Javari2004, an older version of Javari, used mutable fields to represent mutable and assiganble fields. But there wasn't a way to declare a field is only mutable or only assignable. The separation of mutability and assignability makes it possible to declare such fields.

Consider the example:(/*@Q*/ means qualifier @Q is not explicitly written on that position but will be treated as if they are there)

```
class C {
  public mutable /*this-assignable*/ List<String> logs;
  public /*this-mutable*/ /*this-assignable*/ D d2;
  public /*this-mutable*/ assignable D d3;
  public /*this-mutable*/ final D d4 = new D();
}
class D {
  public /*readonly*/ /*this-assignable*/ int x = 0;
}

mutable C c1;
c1.d2 = anotherD;// allowed. d2 is assignable
c1.d2.x++;// allowed. d2 is mutable
c1.d4 = anotherD; // error. d4 is final
```

```
readonly C c2;
c2.logs.add("hi");// allowed. logs is always mutable
c2.d2 = anotherD; // error. d2 is final
c2.d2.x++; // error. d2 is readonly
c2.d3 = anotherD; // allowed. d3 is assignable
c2.d4 = anotherD; // error. d4 is final
```

Re-assignment to `c1.d2` on line 12 is allowed, because `c1` is a mutable reference, `d2` is `this-assignable`, so the assignability of the field `d2` is `assignable`, and therefore allows re-assignments; `c1.d2.x++` is allowed, because `c1` is `mutable` and `d2` is `this-mutable`, mutability of `c1.d2` is `mutable`. `c1.d2.x++` is mutation through a `mutable` reference therefore allowed. Re-assignment to `c1.d4` is forbidden by Java, because `d4` is `final`.

`c2` is a `readonly` reference, therefore mutation to the abstract state of the referred-to object by the `readonly` reference is not allowed. `c2.logs` is a `mutable List` of `String`, therefore adding a new `String` element is allowed on line 17. We say `logs` is outside the abstract state of the object referred by `c2`, thus mutation to it is allowed. `c2.d2` is resolved to `final` because `c2` is `readonly`, therefore re-assignment to `d2` is not allowed on line 18. Remember `d2` was allowed re-assignment on line 12 due to the `mutable` receiver `c1`. This is how `this-assignable` works. Similarly, `c2.d2.x++` is not allowed because `c2.d2` is resolved to a `readonly` reference. `c2.d3` is allowed re-assignment even though `c2` is `readonly` because `d3` is declared to be `assignable`, therefore it's outside the abstract state of the object referred by `c2`. `c2.d4` is not allowed re-assignment because `d4` is `final`, and is forbidden by the Java compiler.

By declaring fields to be `mutable` or `assignable`, Javari allows excluding certain fields from the abstract state of the receiver object. Those fields are open to mutation or re-assignments even through `readonly` references.

Javari supports immediate immutability. In Javari, there is only one possibility: every newly created object is immediately `mutable` after its constructor finishes. For `mutable` objects, delayed immutability doesn't really matter, because a `mutable` reference is always allowed to mutate the object. This only becomes a problem in object-level immutability, when circular references between immutable objects need to be created.

Every newly created object in Javari is `mutable`. So, in order to make an object effectively immutable, every reference to the `mutable` object should be readonly. This definite enumeration of all references isn't scalable, and is not checked by machine, so it's prone to errors and not suitable for beginners.

2.6.2 ReIm And ReImInfer

ReIm[20] is another reference immutability type system. It has three qualifiers: `mutable`, `polyread` and `readonly`. The meanings of these three qualifiers are:

1) `mutable`: a ™reference can be used to mutate the referenced object. The default qualifier for references other than instance fields. ReIm supports concrete immutability, thus it doesn't allow excluding fields from the state of an object. Therefore, `mutable` is forbidden on instance fields to avoid breaking type soundness (because `mutable` fields can be arbitrarily mutated by any kinds of receivers).

2) `readonly`: a `readonly` reference cannot be used to mutate the referenced object, or the transitively reachable objects from it.

3) `polyread`: a polymorphic qualifier. In the direct context where `polyread` is declared, it's effectively `readonly` because the direct context should typecheck whatever `mutable` or `readonly` is substituted in the positions of `polyread`.

ReIm also uses viewpoint adaptations for field accesses and method invocations. Viewpoint adaptation rules for ReIm are:

_ ▷ `mutable` = `mutable`

_ ▷ `readonly` = `readonly`

q ▷ `polyread` = q

For field accesses, ReIm uses receiver contexts as the viewpoint adaptation target, which is the same as the standard viewpoint adaptation. ReIm introduces a concept called calling-context sensitivity. It's basically a variant of viewpoint adaptation: for method invocations, instead of viewpoint adapting declared types of method return types, formal parameters and declared receivers to actual receivers, ReIm adapts them to the calling context, namely the left-hand-side of assignments. It's claimed that the purpose of this is to generalize the viewpoint adaptation: instead of only receivers can be contexts, left-hand-side of assignments can also be contexts. And they call this dependency of viewpoint adaptation on the calling contexts call-transmitted dependencies.

We'll show how this variant of viewpoint adaptation work using the example from *Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity*[20]:

```
class DataCell {
  polyread Date date;
  polyread Date getDate(polyread DateCell this) {
    return this.date;
  }
  void cellSetHours(mutable DateCell this) {
    mutable Date md = this.getDate();
    md.setHours(1);
  }
  void cellGetHours(readonly DateCell this) {
    readonly Date rd = this.getDate();
    int hour = rd.getHours();
  }
}
```

The getDate() method's declared return type is `polyread`. There are two contexts from which getDate() is called — one in cellSetHours() and one in cellGetHours(). In cellSetHours(), the calling context is `mutable Date md`, so the return type of getDate() gets adapted to `mutable` and yields `mutable`, so `mutable Date` is returned. In the calling context, the returned `md` is used to mutate hours. In cellGetHours(), however, the calling context is `readonly Date rd`, so the `polyread` return type of getDate() is adapted to `readonly` and yields `readonly`, so the assignment typechecks. The returned reference isn't used to modify the referenced object, which is consistent with the `readonly` calling context.

Although ReIm is much simpler than Javari, it loses the expressiveness of Javari. First, abstract immutability isn't enforced by ReIm, so every field must be part of the state of an object. This restricted the use of ReIm in real world applications. Second, ReIm has a special viewpoint adaptation for method invocations, which is not consistent with field accesses. For method invocations, the calling context is used as the context, rather than the receiver. The Uniform Access Principle (UAP) states that "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation"[14]. This principle poses restrictions on the syntax of programming languages — there should be no syntactical difference between working with an attribute, pre-computed property, or method/query of an object. To achieve this, field accesses and method invocations should have consistent viewpoint adaptation target — either only receivers or only calling contexts. ReIm's usage of different contexts makes understanding the system very hard. Third, ReIm doesn't support generics — there is no way of specifying, for example, a `mutable List` of `readonly Object` or a `readonly List` of `mutable Object`. Supporting generics is a very important feature that Java programs need. So, ReIm has several limitations that restricted its applicability to real world needs.

ReImInfer is the inference system for ReIm. It uses a set-based approach to infer solutions for references including method formal parameters, declared method receivers, method returns, fields and local variables. Its basic idea is: start from an initial set of qualifiers that make sense for each kind of reference. For partially annotated programs, the set is singleton — it only contains the explicitly written qualifier. ReImInfer keeps refining the set by removing qualifiers that violate type rules of ReIm until: 1) one empty set is reached, meaning there is no solution for the program, or 2) a fix-point is reached in which every set of each reference doesn't change. If any set contains more than one qualifier, ReImInfer chooses one in this order:

`readonly` > `polyread` > `mutable`

ReImInfer is claimed to run at O(n) in practice, but its worst-cast running time is $O(n^2)$.

2.6.3 Google Error Prone Immutable

Google Error Prone[8] is a static analysis tool that catches common programming errors at compile-time. It's used in Google's Guava library. Guava is a set of core libraries that includes new collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection, string processing and so on[5]. Error Prone also has a type declaration annotation @Immutable, to be used on class or interface declarations. Error Prone validates that @Immutable classses and interfaces are deeply immutable by the below conservative definition of immutable:

- All fields are `final`.
- All reference fields are of `immutable` types, or `null`.
- It is properly constructed (the `this` reference does not escape the constructor).
- All subclasses or subinterfaces of `immutable` super classes or super interfaces should also be `immutable`.

All fields being `final` avoids mutation by forbidding re-assignments of fields after an object is initialized. All reference fields being `immutable` makes the immutability guarantee deep. Forbidding `this` to escape from constructors avoids aliases from mutating objects. A class is `immutable` only when all of its instances are `immutable` — an `immutable` class must be either `final` or all the subclasses of `immutable` class should also be `immutable`.

Google Error Prone Immutability is simple and easy to understand, however there are still some big limitations that affect its flexibility:

- It only supports class-level immutability. There is no way to specify that certain instances of a class are immutable, while the others are mutable.
- By making all fields `final`, Java would restrict every field of an `immutable` class to be initialized inside constructors. Calling a helper method that initializes fields, such as init(), within the constructor would make the class ineligible for being `immutable`.
- It's too restrictive to forbid `this` reference from escaping the constructor. For circular initialization of two immutable objects, it's impossible to achieve that as it's not possible for either one object to capture the other object.
- For JDK classes that have non-`final` fields, it's not possible to have `immutable` subclasses.
- There is no way of declaring array types to be @Immutable, because every array is conservatively mutable. Immutable class cannot have fields of array type.

Error Prone's Immutability is concrete immutability, so all objects transitively reachable following the fields of immutable object are inside the state of the object. As a result, it's not possible to support caches. For example, the `String` class from JDK uses a `hash` field to store its hashcode value:

```
@Immutable
public final class String {
  private int hash; // error. not final
  public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
      char val[] = value;
      for (int i = 0; i < value.length; i++) {
        h = 31 * h + val[i];
      }
      hash = h;
    }
    return h;
  }
}
```

If there is @Immutable declaration on class `String`, Error prone would warn the `hash` field in `String` class violates `immutable` classes' specifications. Such classes can't be `immutable` under Error Prone. But from the clients' perspective, `String` class is `immutable` because `hash` field doesn't belong to the abstraction of `String` instances. Error Prone gives us noises when checking similar classes like `String` if they are declared to be @Immutable.

Chapter 3

Context Sensitivity

This chapter is organized as follows: section 3.1 first introduces what context sensitivity is. Section 3.2 discusses receiver-context sensitivity and how it works. Section 3.3 explains assignment-context sensitivity and how it works. Section 3.4 discusses the interactions between the two context sensitivity and how they are coordinated together. Section 3.5 briefly talks about the implementation related information about the two kinds of context sensitivity based on the Checker Framework and Checker Framework Inference.

3.1 Introduction

In context insensitive type systems, when elements are accessed, their types at access sites are the same as their declared types. For example, in the below program:

```
class C {
  @NonNull Object o1 = new Object();
  @Nullable Object o2;
}
...
C c;
c.o1 = null; // error.
c.o2 = null; // ok
```

The type of `c.o1` is the same as the declared type of `o1`, which is @NonNull, so it does not make sense to assign `null` to non-null variable. Similarly, the second assignment is allowed, because `c.o2` has the same type as `o2`'s declared type, @Nullable.

However, some type systems have special semantics such that the types of accesses of elements need to be different from the declared types of the elements. The type of accessed element is the end result of combining types of actual access sites with declared types. We call this context sensitive type refinement, meaning refining declared types according to contexts.

There are two kinds of contexts introduced in this thesis: receiver contexts and assignment contexts. Different contexts have different ways of resolving declared types to yield result types.

3.2 Receiver-context Sensitivity

In receiver-context sensitive type systems, declared types are sensitive to receiver types and resolved using a viewpoint adaptation operation ($\triangleright$)[25, 19]. The viewpoint adaptation operation takes two inputs and yields a single result type. On the left side of the operator is the receiver type, while on the right side is the declared type. The result of viewpoint adaptation is the result type. We then say that the result type is the result of viewpoint adapting the declared type to the receiver type. Viewpoint adaptation is order-sensitive to receiver types and declared types, so switching their positions means a completely different adaptation. Any receiver-context sensitive type system should define a set of rules that specify what the result type is for every combination of qualifiers, which are the so-called viewpoint adaptation rules.

Receiver-context sensitivity is usually applicable to type systems in which there is an ownership relationship between objects and receiver/owner objects' types have an influence on the types of objects accessed through the receiver/owner objects.

In GUT[25, 19], for example, accessing members through references needs viewpoint adaptation as we discussed earlier. For example:

```
public class Person {
  peer Person spouse;
  ...
}
rep Person tony;
tony.spouse; // Has type rep ▷ peer = rep
```

The type of `tony.spouse` isn't the declared type of `spouse` (`peer`). Instead, `tony.spouse` has type `rep`, which is the result of viewpoint adapting the declared type of `spouse` `peer` to the receiver type `rep`. This makes sense, because if the current receiver object owns `tony`, then `tony`'s `peer` `spouse` is also owned by current receiver object, according the semantics of `peer`. This is the semantics of the GUT type system, such that ownership information is correctly transmitted when accessing members.

In PICO, there is also a similar scenario, when an object's mutability should be dependent on its receiver object. One example (`RDM` is the abbreviation of @ReceiverDependentMutable, which implements receiver-context sensitivity in PICO):

```
class Hobby {
  String name;
}
class Person {
  RDM Hobby hobby;
}
immutable Person tony;
tony.hobby.name = "basketball";// error. tony.hobby has type
   immutable ▷ RDM = immutable. Mutation is not allowed
```

Viewpoint adapting `hobby`'s declared type `RDM` to the receiver type `immutable` yields `immutable` according to PICO's viewpoint adaptation rules. Since `tony.hobby` is `immutable`, mutation to `tony.hobby` by re-assigning `name` field is not allowed at line 8.

The above two are both examples of receiver-context sensitivity. Result types are determined by viewpoint adapting declared types to receiver types.

In receiver-context sensitive type systems, viewpoint adaptation happens on four types of accesses of elements (for constructors and methods accesses, they can also be called constructor/method invocations):

- Field Access: accesses of instance field elements. It's already shown in the previous two examples.
- Constructor Invocation: invocations of constructor elements. The constructor's return type and parameter types should be adapted. The constructor's signature after viewpoint adaption should be used as the signature to check whether constructor invocation typechecks or not (in inference, to generate subtype constraints over the viewpoint adapted signature). Receiver type in this case is the type of the new expression.
- Method Invocation: invocations of instance methods. The method return, the declared receiver and parameter types should be viewpoint adapted to the receiver type. Typechecking happens against the viewpoint adapted signature of the method (in inference, similar to constructor invocation scenario).
- Instantiation of Type Parameters: lower and upper bounds of type parameter should be adapted to the receive type. The type argument should be within the adapted lower bound and adapted upper bound of the type parameter.

Since receiver-context sensitivity always needs receivers, there is no viewpoint adaptation happening on accesses of static elements such as static field accesses and static method invocations, as there are never receivers in those cases.

3.3 Assignment-context sensitivity

Assignment-context sensitive type refinement is inspired by Java's generic methods. In Java, generic methods can declare method type parameters, and use them on method return types and formal parameters. When the generic method is invoked, the actual signature of the method is resolved based on the types of assignment context and arguments. However, in Java, method declared receiver can't be a type parameter, because a method must be declared in non-type-parameter class and that class should be the declared receiver type. Whether the invocation of the generic method typechecks or not depends on if there exist valid substitutions for all the method type parameters. If there are such substitutions, the invocation typechecks. Otherwise, it doesn't. To illustrate, one example is:

```
class C {
  <T> T foo(T t) {
    return t;
  }
}
C c;
Object o = c.foo("Hello"); // ok
```

```
String s = c.foo(new Object); // error
```

The constraints for inferring a solution for the invocation of the generic method foo() on line 7 are as follows:

`String` <: `T` (on method argument passing)

`T` <: `Object` (assignment of method return to assignment context)

T has two solutions, `String` and `Object`, so the method invocation one line 7 typechecks. However, for the second method invocation on line 8, the generated constraints are:

`Object` <: `T` (on method argument passing)

`T` <: `String` (assignment of method return to assignment context)

There isn't any solution for `T`, therefore, this invocation on line 8 doesn't typecheck.

Compared to Java, pluggable type systems use qualifiers as extended types on method declared receiver types. Since they don't affect the resolution of the class to which a generic method belongs to, assignment-context sensitive qualifiers can be used on method declared receiver just like they can be used on formal parameters. The assignment-context sensitivity we'll discuss later refers to this more flexible version that can be applied also to declared method receivers.

Similar to Java, the resolution of an assignment-context sensitive method's signature in terms of extended qualifiers on invocation sites is as follows: based on the actual receiver, arguments and assignment context's types, whether there is any valid substitution such that after replacing the assignment-context sensitive qualifiers on the method, the invocation typechecks. For example, in PICO:

```
class A {
  @PolyMutable Object foo(@PolyMutable A this, @PolyMutable
     Object p) {...}
}

@Mutable A a;
@Mutable Object mo = a.foo(new @Mutable Object()); // OK
@Immutable Object imo = a.foo(new @Immutable Object()); // error
```

Method invocation on line 6 has solution @Mutable, therefore it typechecks. However, the second invocation generates these constraints[1]:

`@Mutable` <: `@PolyMutable` (on method receiver)

`@Immutable` <: `@PolyMutable` (on argument passing)

`@PolyMutable` <: `@Immutable` (assignment of method to assignment context)

According to the PICO mutability qualifier hierarchy (section 4.2.1), there are no solutions, so the invocation on line 7 doesn't typecheck.

[1]Actually, @SubstitutablePolyMutable is used in the place of @PolyMutable. But we haven't explained their differences yet. For better explanation purpose, we still use @PolyMutable here. We'll discuss the reason of this in chapter 4.

Compared to the receiver-context-sensitive type refinement, assignment-context-sensitive type refinement only applies to method invocations (some type systems may allow constructor invocations too), but not on field accesses or type parameter instantiations.

PICO has a dedicated qualifier, @PolyMutable, which is similar to method type parameter `T` in generic methods in Java, to represent mutability types that should look for a valid substitution. It implements assignment-context sensitivity in PICO. Using only receiver-context-sensitive type refinement doesn't consider the assignment context's effect and limits expressiveness in some cases, such as the factory design pattern. In PICO, for example:

```
1  @Immutable
   class ImmutableFactory {
3    @ReceiverDependentMutable Object getObject1(@Readonly
        ImmutableFactory this) {
       return new @ReceiverDependentMutable Object();
5    }
     @PolyMutable Object getObject2(@Readonly ImmutableFactory this
        ) {
7      return new @PolyMutable Object();
     }
9  }

11 class Test {
     @Immutable ImmutableFactory imfactory = ...;
13   void test1() {
       @Immutable Object oim = imfactory.getObject1(); // ok
15     @Mutable Object om = imfactory.getObject1(); // error.
          return type is @Immutable
     }
17   void test2() {
       @Immutable Object oim = imfactory.getObject2(); // ok
19     @Mutable Object om = imfactory.getObject2(); // ok
     }
21 }
```

From the above example, we can see that the return type of `ImmutableFactory#getObject1()` is declared as @ReceiverDependentMutable Object. However, all instances of the class ImmutableFactory are `immutable` because there is an @Immutable annotation on the declaration of ImmutableFactory. This essentially means that `ImmutableFactory#getObject1()` can only return `immutable` objects back, and can never create a @Mutable Object. This doesn't make sense, because conceptually, the created object shouldn't be part of the abstract state of the factory. An @Immutable factory should be able to create objects with any mutability types: `mutable` or `immutable`. One possible solution is to create two copies of the method `ImmutableFactory#getObject1()`, of which one returns @Immutable Object, and the other returns @Mutable Object, but Java's method signature doesn't consider annotations on method declarations, so those two methods will be considered the same, and rejected. This approach also duplicates code, which is not a nice way to solve this problem.

Using assignment-context sensitivity, however, the return type of `ImmutableFactory#getObjec`

t2() is resolved by assignment context only in this example, without the influence of actual receiver (if the declared receiver of this method also has @PolyMutable, @PolyMutable resolution is also affected by actual receiver, but here, @Readonly is used). In the third method invocation on line 18, @PolyMutable has inferred solution @Immutable, so the method invocation typechecks. In the last method invocation on line 19, the return type of `ImmutableFactory#getObject2()` is resolved to @Mutable, so the invocation also typechecks. Assignment-context sensitivity effectively solves the problem. By having assignment-context sensitivity, we can reduce code duplication when a method return type should be sensitive to the assignment context.

Assignment context can be a real assignment to a variable or can be pseudo-assignments such as being used as the actual receiver for another method invocation, passed as arguments to another method invocation etc. For example, for method invocation `a.b()`:

```
1 @Mutable Object o = a.b();// Assignment context is @Mutable
     Object
  a.b().c();// Assignment context is c()'s declared receiver type
3 d.e(a.b());// Assignment context is e()'s formal parameter type
```

To emphasize the ability of being sensitive to assignment contexts, we call this kind of context sensitivity assignment-context sensitivity, even though one can declare a method that doesn't have assignment-context sensitive qualifier (such as @PolyMutable in PICO) on the method return. In this case, the assignment contexts' types have no influence on the method's signature resolution.

3.4 Interaction Between Receiver-Context Sensitivity And Assignment-Context Sensitivity

If a type system supports both receiver-context sensitivity and assignment-context sensitivity, there is one scenario that needs attention: if viewpoint adapted signatures of methods can contain assignment-context sensitive qualifiers, then those adapted assignment-context sensitive qualifiers shouldn't take part in the resolutions. Only those assignment-context sensitive qualifiers that are declared directly on method signatures should be resolved. We'll show how this is handled in PICO using an example in section 4.8.14.

3.5 Implementation Details

3.5.1 Receiver-context sensitivity

To implement receiver-context sensitivity as a framework-level feature, we add a new class ViewpointAdapter that has the abstract logic of traversing over different types and combining types during viewpoint adaptation. Previously this logic was inside GUTUtils. We generalized and extracted it from GUTUtils to ViewpointAdapter. ViewpointAdapter is used in AnnotatedTypeFactory in the Checker Framework to perform viewpoint adaptation if there is a non-null viewpoint-adapter instance provided by type system developer. In the four locations that should have viewpoint adaptation, AnnotatedTypeFactory uses ViewpointAdapter to perform a standard implementation of

viewpoint adaptations. Subclasses of AnnotatedTypeFactory can override corresponding methods for each location to have different behaviours specific to type systems. In the Checker Framework Inference, there is an InferenceViewpointAdapter that extends ViewpointAdapter, and overrides necessary methods to generate CombineConstraints between declared, receiver and result slots.

For typechecking, a type system only needs to provide a concrete implementation of FrameworkViewpointAdapter, which is an abstract subclass of ViewpointAdapter in the Checker Framework side, to specify the result types for each combinations of types. That's everything a type system developer needs to let their type systems to support viewpoint adaptation.

On inference side, type system developers in most cases don't need to override InferenceViewpointAdapter, as the default implementation is already able to handle constraints generation for viewpoint adaptation. If a type system has simpler viewpoint adaptation rules, they can override InferenceViewpointAdapter to improve the performance, but other than that use case, there's no need to override InferenceViewpointAdapter. However, type system developers do need to provide solver encoding logic for CombineConstraints.

3.5.2 Assignment-context sensitivity

There is a meta-annotation @PolymorphicQualifierInHierarchy in the Checker Framework. It can only be applied to method returns, formal parameters and method declared receivers. Resolution of it is quite similar to Java's generic method resolutions, except the assginment context has no effect. This is because in the previous implementation of resolution algorithm, QualifierPolymorphism, only arguments and actual receivers' types were considered and took part in the resolution process. We extend the existing logic of QualifierPolymorphism to also consider assignment context with the help of TypeArgInferenceUtil. This class is a utility class that can find types of direct assginments or peudo-assignments.

Chapter 4

Practical Immutability For Classes And Objects (PICO) Type System

This chapter is organized as follows: section 4.1 gives an overview on PICO type system. Section 4.2 introduces the qualifier hierarchy of PICO. Section 4.3 introduces viewpoint adaptation rules for PICO. Section 4.4 gives several examples that motivate the key features of PICO. Section 4.5 talks about what kinds of fields belong to the abstract state of the enclosing object. Section 4.6 discusses implicitly immutable types in PICO. Section 4.7 explains how defaults are performed for unannotated locations when no implicit types apply. Section 4.8 explains language design of PICO with small examples that motivate each type rule. Finally, section 4.9 formalizes PICO type system without generics.

4.1 Overview

Immutability has many benefits such as safe sharing of objects between threads, more reliable software, better understanding and reasoning about software behaviors, etc. In this chapter, we present a new immutability type system called PICO.

PICO is a static type system that extends Java's standard type system to additionally enforce object immutability guarantees for Java programs. In PICO, objects on the heap are either mutable or immutable. Immutable objects' abstract state is never changed once they are allocated and fully initialized. PICO also supports reference immutability. Among references, readonly reference is the most interesting — it can point to either mutable or immutable objects, and only perform read operations, without needing to know the concrete mutability information of the underlying data structure.

PICO is sensitive to both the receiver context and the assignment context using the @ReceiverDependentMutable and @PolyMutable qualifiers respectively. With the receiver-context sensitivity, a field, a method, or a class is able to be instantiated to either mutable or immutable types, depending on receivers. Using the assignment-context sensitivity, PICO supports methods whose mutability can depend on assignment contexts.

PICO's immutability guarantee is over the abstract state of objects. Only objects that are part of the abstraction of the receiver are under the immutability guarantee. Objects are still

allowed to have mutable in-memory representations not visible to clients, such as those for caching. Assignability and mutability of fields are decoupled and enforced independently, which greatly increases the space of possible kinds of fields. They together determine whether a field belongs to the abstract state of the receiver. It's up to end-users' decision to declare fields to reflect their needs and determine whether they should be under immutability protection or not.

PICO transitively protects every object that belongs to the abstract state of the receiver. We believe that deep immutability is more useful to users, as it can provide stronger guarantees about the behavior of the program thus making it easier to analyze and reason about code.

Initialization of immutable objects is a big challenge for object immutability type system designers. Every object including immutable objects starts as being mutable, such that they can be initialized. PICO leverages an existing initialization-tracking type system, Freedom Before Committment[23], and combines it with mutability qualifiers to guarantee initialization is properly performed for immutable objects during initialization phase. Circular initialization of immutable data structures is also supported, to instantiate mutually dependent data structures.

Because PICO is a static type system, it doesn't add any runtime overhead to the bytecode of compiled programs, because at runtime, mutability qualifiers will be erased. Programs that typecheck under PICO without programming constructs that need runtime enforcement such as downcasts, will definitely satisfy the immutability guarantee during execution. However, if mutability qualifiers were reserved at runtime, the JVM had the opportunity of improving the performance based on the mutability information, even though lacking ways to utilize pluggable types at runtime is a general problem for every pluggable type system based on the Checker Framework right now.

4.2 Qualifiers And Hierarchies

PICO has two orthogonal qualifier hierarchies. One is for initializations, which is exactly the same as Freedom Before Commitment type system's qualifier hierarchy. The other is the mutability hierarchy. There is third dimension, assignability qualifiers, but they are not part of the type. Assignability qualifiers are just declarations on fields to specify if a field is allowed to be assigned or not. Two types are subtypes only when initialization qualifiers, mutability qualifiers and Java types all satisfy their subtype relations.

4.2.1 Mutability Hierarchy

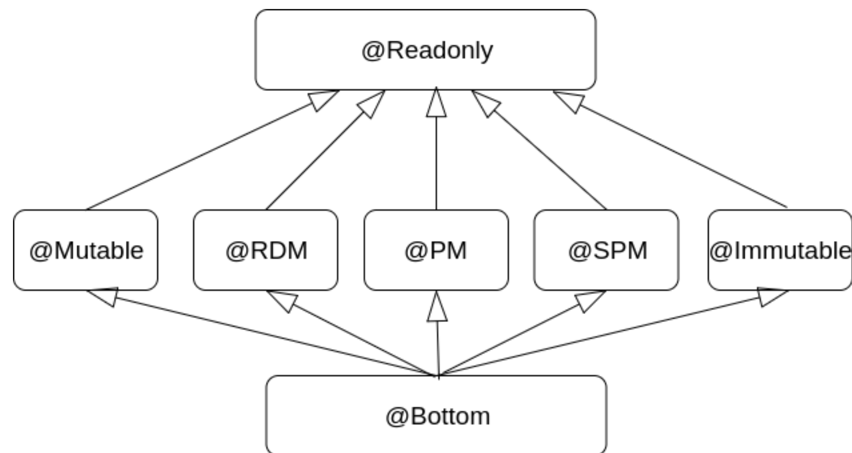

Figure 4.1: Qualifier hierarchy of mutability

The meaning of each qualifier is:

- @Readonly: only applied to references. A `readonly` reference can never be used to modify the abstract state of the referred-to object.

- @Mutable: applies to references, object creations and type element declarations (type element refers to classes and interfaces declarations in Java). A `mutable` object can be mutated at anytime after it is allocated on the heap. A `mutable` reference can only refer to mutable objects.

- @Immutable: applies to references, object creations and type element declarations. `immutabl e` objects are closed to modifications. Once it's allocated and fully initialized, its abstract state can never change in the entire lifetime. `immutable` references can only refer to `immutable` objects.

- @ReceiverDependentMutable (@RDM): applies to references, object creations and type element declarations. We'll use `RDM` as the shorthand for `receiver-dependent-mutable`. `RDM` objects are special in that their mutability is the same as the actual receiver's mutability. The receiver-context sensitivity of PICO comes from this qualifier.

- @PolyMutable (@PM): applies to both references and object creations. Only used on method return, formal parameters and declared method receiver, or on object creations. Its mutability is inferred depending on method arguments, the actual receiver and the assignment-context type on invocation sites. The assignment-context sensitivity of PICO comes from it.

- @SubstitutablePolyMutable (@SPM): internal qualifier that is invisible to clients. Can never be written by users. When a method with at least one @PolyMutable is invoked, all occurences of @PolyMutable will be transformed to a temporary @SubstitutablePolyMutable first, then substituted by one actual mutability solution.

- @Bottom: represents the bottom type. Can only be used for the type of the `null` literal and lower bound of type variables.

Note: in the remainder of the thesis, by saying "type", we really mean "mutability types", and they never include `substitutablepolymutable` and `bottom` except as explicitly mentioned.

4.2.2 Assignability Dimension

There are three assignability qualifiers:

- `final`: used on references. `final` fields are fields that cannot be re-assigned. It is enforced by Java. `final` fields must be assigned values in constructors. Java additionally allows using `final` on references other than fields such as method parameters, but it doesn't affect how PICO handles assignability. PICO never warns on re-assignments to references other than fields if they are not declared with `final`. If they are `final`, then it's enforced by Java.
- @Assignable: used on fields. @Assignable is the opposite of `final`, meaning that a field can always be re-assigned.
- @ReceiverDependentAssignable (@RDA)[1]: only used on instance fields. We'll use `RDA` to be the shorthand to represent `receiver-dependent-assignable` in the remainder of the thesis. Assignability of a `RDA` field is determined solely by the actual receiver: if the actual receiver is `mutable`, then this instance field can be re-assigned. Otherwise, the field cannot be re-assigned as if it's declared to be `final`.

Assignability qualifiers don't belong to types. They are only declared qualifiers that specify restrictions on the assignability of a field. The extended types (to Java types) contain one and only one initialization qualifier and mutability qualifier respectively.

4.3 Viewpoint Adaptation Rule

PICO performs viewpoint adaptation for only the mutability qualifier hierarchy, not for the initialization hierarchy, because initialization qualifiers are not receiver-context sensitive. The viewpoint rules for combining mutability qualifiers are (q represents any mutability qualifier):

$q \triangleright \texttt{RDM} = q$
$_ \triangleright q = q$ (otherwise)

From the above rule, we can find that only `RDM` is sensitive to the receiver context. It is a "transparent" qualifier such that the result type is the same as the receiver's type. Other qualifiers are "opaque": their mutability information is well-known, fixed, and doesn't depend on the receiver's type.

[1] No writable qualifier exists for @ReceiverDependentAssignable for conciseness. If `final` and @Assignable don't apply, it's equivalent to using `receiver-dependent-assignable`

4.4 Motivating Examples

We use several examples to illustrate and motivate some key characteristics of PICO. Again, we use block comments (/*@Q*/) to represent that the qualifier @Q is not explicitly written in the source code — they are either implicitly applied or defaulted to the location.

4.4.1 Immutable Object And Its Creation

```
1 Person imp = new /*@Immutable*/ Person();
  imp.age++; // error. imp is immutable
3 @Readonly Person rop = imp;
  rop.age++; // error. rop is readonly, not allowed to mutate
     referent
```

The @Immutable annotation on the `Person` class declaration means all its instances are immutable. `new Person()` is implicitly applied `immutable` type, because the `Person` class is declared with @Immutable. `imp` points to an `immutable` object, whose abstract state cannot be changed. After being assigned to `rop` reference, the `immutable` object behind cannot be changed, either.

4.4.2 Receiver-Context Sensitivity

We can declare methods that are receiver-context sensitive using the qualifier @ReceiverDependentMutable. The @ReceiverDependentMutable annotation on the `Planet` class declaration means that it can be instantiated as both `mutable` or `immutable` instances.

```
  @ReceiverDependentMutable
2 class Planet {
    /*@ReceiverDependentMutable*/ List</*@Immutable*/ String>
       aliases = ...;
4 }
  ...
6 @Mutable Planet mp = ...;
  mp.aliases.add("Earth"); // OK, since mp is @Mutable, mp.aliases
      is also @Mutable (@Mutable ▷ @RDM = @Mutable)
8 @Immutable Planet imp = ...;
  imp.aliases.add("Jupiter"); // error. imp is @Immutable, imp.
     aliases is @Immutable
```

The root reason of having `RDM` is the transitive guarantees PICO provides: `mutable` objects have `mutable` fields, `immutable` objects have `immutable` fields, and `readonly` references also have only `readonly` access to the referred-to object. It's necessary to have a uniform qualifier that works in all kinds of receiver type cases. Therefore, we have such a qualifier, and we name it @ReceiverDependentMutable, to reflect the fact that its real type is determined and dependent on the receiver's type.

4.4.3 Assignment-Context sensitivity

```
class ProductFactory {
  @PolyMutable Product createProduct() {
    return new @PolyMutable Product();
  }
}
...
@Mutable ProductFactory factory = ...;
@Mutable Product mp = factory.createProduct(); // OK
@Immutable Product imp = factory.createProduct(); // OK
```

The createProduct() methods's return type is declared to be @PolyMutable. Both invocations of the method typecheck. The returned Product's mutability completely depends on the assignment context: if the assignment context expects a @Mutable Product, a @Mutable object is returned; the same argument holds for an immutable assignment context. The returned object's mutability is independent from the receiver's mutability.

4.4.4 Separation of Assignability and Mutability

```
class C {
  final /*@ReceiverDependentMutable*/ Person p1;
  /*@ReceiverDependentAssignable*/ @Immutable Person p2;
}
...
@Mutable C c = ...;
c.p1 = another Person; // error. Not allowed to be re-assigned
c.p1.age = 25; // OK. c.p1 is @Mutable
c.p2 = another Person; // OK. c.p2 is @Assignable
c.p2.age = 30; // error. c.p2 is @Immutable.
```

A field's assignability and mutability are independent. A field can be `mutable`, allowing calling side-effecting methods on it, but forbidden to be re-assigned to point to a new object; a field can be `immutable` itself, ensuring that the referred-to object won't get corrupted, but allowing the field to point to another `immutable` object.

4.4.5 Transitive Immutability Guarantee

```
@ReceiverDependentMutable
class A {
  /*@ReceiverDependentAssignable*/ /*@Immutable*/ int x = 0;
}
@ReceiverDependentMutable
class B {
  /*@ReceiverDependentAssignable*/ /*@ReceiverDependentMutable*/
      A a = new @ReceiverDependentMutable A();
```

```
}
@Immutable
class C {
  /*@ReceiverDependentAssignable*/ /*@ReceiverDependentMutable*/
      B b = new @Immutable B();
}
...
@Immutable C c = ...;
// Mutation to c
c.b = another B; // error. c is @Immutable, b is resolved to
   final
// Mutation to c.b (direct field of c)
c.b.a = another A; // error. c.b is transitively @Immutable, a
   is resolved to final
// Mutation to c.b.a (second level field of c)
c.b.a.x++;// error. c.b.a is transitively @Immutable, x is
   resolved to final
```

We can see for @Immutable objects, PICO not only forbids mutation to the immutable object itself, in this case `c`, but also forbids mutation to the direct fields `c.b`, fields of fields `c.b.a`. Therefore, PICO does provide the transitive immutability guarantees such that all objects in abstract state that are transitively reachable from the root immutable object are also immutable.

4.4.6 Exclude Fields From Abstract State

In the previous examples, all the fields are protected under the immutability guarantee: if the receiver object is @Immutable, those fields cannot be re-assigned to point to a new object (or store new values in primitive types cases). Side-effecting methods are also forbidden to be called on those fields (not shown in the examples). However, users can break this restriction by declaring a field to be @Assignable, @Mutable or @Readonly. By this, a field will be excluded from the abstract state of an object, so that PICO will still consider the receiver object unchanged, even if that field is re-assigned, or the referred-to object is mutated.

Assignable field example:

```
@Immutable
class CharContainer {
  private @Assignable /*@Immutable*/ int hc = 0;
  char[] data;
  int hashCode(/*@Readonly*/ CharContainer this) {
    if(hc == 0 && data.legnth != 0) {
      // OK to re-assign hc even though "this" is @Readonly
      hc = some computations on data field;
    }
    return hc;
  }
}
```

The hashCode() method is implicitly declared to be @Readonly — in the method body, normally it's not allowed to mutate the current receiver object `this`. However, by making the `hc` field `assignable`, re-assignment to `hc` is considered to be a benevolent side effect. The belief behind this is that, since a field can be re-assigned at any time, it shouldn't be part of the abstraction of the receiver. Otherwise, it means programmers must have misunderstood their program to let the abstraction of a class depend on an always-assignable field. By marking fields to @Assignable or @Mutable/@Readonly, they will be excluded from abstract state. A `mutable` field can be mutated by arbitrary clients that hold the reference to it, therefore it's out of the owner object's abstract state. `readonly` fields are outside the abstract state because the underlying data structure may be `mutable` so the owner can't guarantee the referred-to object doesn't change. Section 4.5 discusses what kinds of fields belong to the abstract state in detail.

4.4.7 Initialization of Immutable Objects

Immutable objects are effectively "mutable" inside their constructors so that fields of them can be initialized. To keep track of the initialization state of immutable objects, so that they are initializable, PICO leverages Freedom Before Commitment type system.

```
@Immutable
class Car {
  /*@ReceiverDependentAssiganble*/ /*Immutable*/ int
     numberOfWheels;
  /*@Immutable*/ Car(int numberOfWheels) {
    this.numberOfWheels = numberOfWheels;
  }
}
```

`this` in the `Car` constructor has type @Immutable. In previous examples, @Immutable receivers are not allowed to assign fields. However, in the construction phase of immutable objects, assignments via immutable reference are allowed. Actually, here is the point to introduce the initialization qualifier hierarchy, which we didn't mention explicitly in the previous examples. `this` in the `Car` constructor actually has type @UnderInitialization @Immutable. This type indicates that the object is currently under initialization, so even though it points to an `immutable` object, assignments are allowed. `immutable` receivers mentioned earlier all have type @Initialized @Immutable, meaning the `immutable` object is already fully initialized, thus any mutation to that object is forbidden.

Accroding to the FBC type rules, once the `Car` constructor finishes, the newly created instance is considered to be fully initialized and have type @Initialized @Immutable. Thus, the mutation at line 2 isn't allowed.

```
@Initialized @Immutable Car imcar = new @Immutable Car(4);
imcar.numberOfWheels++; // error. imcar is @Initialized
   @Immutable, numberOfWheels is resolved to final
```

4.5 Abstract State

Since PICO is an abstract immutability type system, it is important to know what fields belong to the abstract state so that they can be correctly declared. Below is a figure that illustrates this:

Assignability \ Mutability	mutable	readonly	immutable	RDM
assignable	x	x	x	x
final	x	x	✓	✓
rda	x	x	✓	✓

Figure 4.2: Field declarations and abstract state

Figure 4.2 illustrates what combinations of assignability qualifiers and mutability qualifiers for fields are part of the abstract state of the enclosing object. x means the corresponding field is not in the abstract state, while ✓means the field is in the abstract state.

`mutable` and `readonly` fields are not part of the abstract state of the enclosing object. The reason is that `mutable` fields can always be mutated by any alias, which is completely out of the control of the enclosing object. So, `mutable` fields are excluded from the abstract state. `readonly` fields are outside the abstract state, because the referenced objects may be `mutable` objects, as `readonly` is the top qualifier that can be assigned any objects with any mutability. Enclosing objects have no knowledge of the underlying objects stored to ensure anything about them. A `readonly` field is useful to pass a data structure to a particular class to only read information, but not write to it nor care about if the data structure is `mutable` or not.

Every `assignable` field is outside the abstract state, too. This is because `assignable` field can be re-assigned to refer to a different object at any time. So the identity of the fields are not guaranteed to be the same.

Only when a field is `RDA` or `final`, and the mutability type of the field is recursively `RDM` or `immutable`[2], then the field is in the abstract state of the receiver object.

In `immutable` objects, every recursively `RDM` field is resolved to an `immutable` object or a set of `immutable` objects (for example, lists and arrays). `immutable` declared fields are `immutable`, no matter what the receiver is. Therefore, the objects referred to by `RDM` and `immutable` fields can not be mutated via `immutable` receivers. The other concern is assignability of fields. Even though the objects are not `mutable`, if the field is re-assigned to point to another object, the `immutable` object's state also changes. `final` or `RDA` fields prevent this. `RDA` fields will be resolved to `final` in `immutable` receiver objects, so re-assigning them would be forbidden. `final` fields are even safer: Java compiler ensures that `final` fields are only assigned once in constructor, so there is no re-assigning problems for `final` fields.

One interesting situation to consider is generic classes with type parameters, for example:

```
class List<E> {
2   ...
}
```

[2]Mutability requirement here applies recursively to composite types and type parameter uses — for arrays, array dimension types; for parameterized types, type arguments; for type parameter, its declared upper bound.

As we discussed before, if a type parameter's upper bound is recursively either `immutable` or `RDM`, and the assignability of the field is still `final` or `RDA`, then any field that contains the type parameter in its declared type[3] is still in the abstract state.

Let's see an example:

```
@Immutable
class Container<E extends @Immutable Person> {
  E /*@ReceiverDependentAssignable*/ /*@ReceiverDependentMutable
      */ [] elements;
  ...
}
/*@Immutable*/ Container<@Immutable Person> imc = ...;
...some initialization...
imc.elements.add(new @Immutable Person()); // error. elements is
    immutable array
imc.elements[0].age++; // error. Immutable person is stored
   inside the Container
```

Field `elements` belong to the abstract state of the container. Not only the container itself's state is guaranteed to not change (adding/removing elements are not allowed), but also the stored elements are unmodifiable.

More interestingly, users can still exclude `elements` from the abstract state of the enclosing container class by declaring the type parameter's upper bound as @Mutable or @Readonly.

For example, if we have:

```
@Immutable
class ShallowImmutableContainer<E extends @Mutable Person> {
  E /*@ReceiverDependentAssignable*/ /*@ReceiverDependentMutable
      */ [] elements;
  ...
}
ShallowContainer<@Mutable Person> imsc = ...;
...some initialization...
imsc.elements.add(new @Immutable Person()); // error. elements
   is immutable array
imc.elements[0].age++; // OK. Stored Person is mutable.
```

Even though the container itself is ™, the stored elements are still also `mutable`, which are excluded from the immutability guarantee of the container; In such case, we can also say `elements` are outside the abstract state of the shallow `immutable` container.

By properly declaring upper bounds of type parameters, it's up to user's choice to adopt the shallow immutability guarantees or the deep immutability guarantees regarding elements' types.

[3]Maybe directly declared a field with the type parameter type, or used the type parameter in composite types, for example array component type in arrays or type argument for parameterized types

4.6 Implicits

There are Java types whose mutability types are implicit. For example, every String in Java is `immutable`. They cannot be mutated by any means. It would be nonsense to say "@Mutable String". Primitive literals and primitive types are also implicitly `immutable`. For example:

```
int x = 0; // 0 is immutable.
// re-assignment doesn't modify integer literal 0.
// Instead, x simply stores another immutable int value 5
x = 5;
// Integer instance can never change its internal value(Java
   doesn't provide
// methods to do this)
Integer y = new Integer(2);
// Reassignment to y doesn't mutate original Integer instance,
// y only points to a new Integer instance that stores value 3
y = new Integer(3);
```

Right now, these are the implicitly `immutable` types in PICO:

- All literals : primitive (1, 1.0 etc.), string ("hello") and class literals (Object.class etc.). Primitive literals include: integer, short, long, float, double, byte, char literals.
- Reference types: java.lang.Enum, java.lang.String, (java.lang.)Double[4], Boolean, Byte, Character, Float, Integer, Long, Short, Number, BigDecimal, BigInteger.
- Primitive types: int, short, long, float, double, byte, char types. The difference between 1) and 3) is that, 1) applies to literals, and 3) applies to types. Literals and types are treated separately in the Checker Framework, so we also list them separately here.
- Enum classes and enum constants

Implicit types are built-in, and can never be overriden. Otherwise, well-formedness of the type is broken and will be reported as errors.

4.7 Defaults

Not every program is fully or even partially annotated. In order to typecheck these programs, a type system needs a mechanism to apply some default qualifiers from the type hierarchy if there are no explicitly written qualifiers or no implicit type apply. This mechanism is defaulting. Defaulting should be carefully chosen so that it reduces the number of errors and warnings to a reasonable level, yet is still reasonably strong to reflect the real world need. Reducing the number of errors and warnings is not the only goal — defaults should also maximize the guarantees that the type system tries to enforce. If an improper default is chosen, then it's more possible to raise false

[4]prefix java.lang. is omitted. The same for the other reference types in this list.

positive warnings that eventually make type system's error messages less valuable. Or, the program doesn't raise any warnings, but the properties users are interested in may not be guaranteed. So, choosing a proper default is a balance between reducing the number of false positive warnings and maximizing the degree of guarantees over the typechecked programs.

Default qualifiers can be overridden. If there is an explicit qualifier written on a type use, then that explicit qualifier will be used.

In previous sections, we used /*@Q*/ to represent not-explicitly-written qualifiers. However, from this point, we'll use /*@Q*/ uniformly in examples to represent @Q is defaulted to the corresponding location instead of being explicitly written or applied implicitly in the remainder of the thesis.

4.7.1 Initialization Defaults

For initialization qualifiers, if not explicitly written, @Initialized is the default. For extremely rare cases, @UnderInitialization or @UnknownInitialization qualifier needs to be used, for example, calling instance methods from constructors needs the called method's declared receiver to be @UnderInitialization, because `this` in constructors is implicitly @UnderInitialization. @Initialized default can handle the majority of cases.

4.7.2 Mutability Defaults

4.7.2.1 Mutability Default For Type Element Bound

In PICO, there is a mutability qualifier on the type element to restrict its instances' mutability. We also call that mutability annotation the type element bound. By default, a type element bound is `mutable`, meaning all instances of that type element are `mutable`. Users need to explicitly write `RDM` or `immutable` on type elements when the type elements need to be instantiated to either `mutable` or `immutable`, or `immutable` only, respectively.

4.7.2.2 Mutability Default For Usages of Type Element

Usages of type elements are uses of types, which are the opposite to declarations of types. For type elements declared with `mutable`, every usage type is by default `mutable`. For an `immutable` type element's usages, they are defaulted to `immutable`. For an `RDM` type element's usages, they are defaulted to be `mutable`, except on instance fields (they are defaulted to `RDM`).

4.7.2.3 Mutability Default For Fields

Instance fields whose type elements are bounded with `RDM` are defaulted to `RDM`. The reason for having the precondition that the field's type element has to be `RDM` is that it is a violation of type rules to have `RDM` usages of `mutable` or `immutable` type elements: usages of `mutable` or `immutable` type elements on fields are defaulted to the same as their type elements' mutability. In order to maximize the number of objects that belong to the abstract state of the enclosing object, for `RDM` type elements, we default `RDM` to their usages on fields.

Array types are special: they don't have type element declarations. PICO defaults @ReceiverDependentMutable to array types on fields to increase the number of objects that belong to the abstract state of the enclosing object as much as possible.

Static fields have the same defaulting way as usages of type elements. A static field doesn't belong to the state of any object. Instead, it's shared by all the instances of the corresponding class. It doesn't make sense to default static fields into `RDM`, even if the corresponding type element is `RDM` since there are no receivers for them. Actually in static contexts, @ReceiverDependentMutable is forbidden to be used.

4.7.2.4 Mutability Default For Constructor Return And New Expression

Constructor return type restricts the mutability of the instances created by the constructor. By default, a constructor's return type has the same mutability type as its type element bound. For example:

```
@Immutable
class Factory {
  /*@Immutable*/ Factory() {}
}
```

By default, types of new expressions have the same mutability type as the corresponding constructor if the constructor's return type is `mutable` or `immutable`. If the constructor's return type is `RDM`, new expression's type is defaulted to `mutable`.

4.7.2.5 Mutability Default For Local Variable

Local variables are defaulted to be `readonly` so that every mutability-typed object can be assigned to it. However, local variables' types are flow-sensitively refined to the actual type on the right-hand-side of the assignment, and further read operations of that local variable use the refined type to be more accurate. Write operations' validity are still checked against the declared types of local variables. For example:

```
void foo() {
    /*@Readonly*/ Person p = new @Mutable Person();
    p.setName("newName"); // ok. Because p is flow sensitively
       refined to @Mutable
    p = new @Immutable Person(); // allowed. Declared type of p
       is @Readonly
    p.setName("AnotherName"); // error. p is now refined to
       @Immutable
}
```

Flow-sensitive type refinement is a way in the Checker Framework to reduce burden of annotating types and eliminate false positive warnings. Checkers treat local variables and expressions within a method body as having a subtype of their declared or default types. It doesn't introduce unsoundness nor causes an error to be missed[2].

4.7.2.6 Mutability Default For Lower And Upper Bound of Type Parameters and Wildcards

In order to let every possible type argument be passed, `bottom` is used as the default lower bound and `readonly` is used as the default upper bound for a type parameter or a wildcard.

4.7.2.7 Mutability Default For The Other References

If none of the above defaults apply, unannotated references are defaulted to `mutable`. In object-oriented programs, objects' states frequently change, such that they can store data, make computations, interact with each other etc. So, `mutable` best reflects the real case. For functional programming, one can change the default, to `immutable` to achieve functional-programming-like behaviors using an object-oriented style.

4.7.3 Assignability Defaults

For assignability qualifiers, instance fields are defaulted to `RDA`, so that existing re-assignments via defaulted `mutable` references still typecheck, yet it still guarantees more fields in the abstract state (together with the mutability qualifier of course). Static fields are defaulted to `assignable`, if `final` is not used.

4.8 Language Design

4.8.1 Valid New Instance Creation Types

PICO allows 4 kinds of object creation types: `mutable`, `immutable`, `RDM`, and `polymutable`. Creations of `mutable` and `immutable` instances are straightforward. Instances of `RDM` and `polymutable` may not be intuitive. PICO's type rules ensure that their instances' mutability are resolved to either `mutable` or `immutable` according to the corresponding contexts' types, and at runtime there won't be both `mutable` references and `immutable` references to the same object. `readonly` instance creations don't make sense, because in the runtime model, every object should be either `mutable` or `immutable`.

4.8.2 Type Element Bound

Type element bound specifies what kinds of instances can be created from that type element. It can be any of the three among `mutable`, `immutable`, and `RDM`. In the later chapters, we'll use type declarations interchangeably with type elements.

`mutable` classes can only have `mutable` instances. `immutable` classes can only have `immutable` instances. `RDM` classes can have either `mutable`, `immutable`, `RDM` or `polymutable` instances. By "instances", we mean the types of `new` expressions, not the usage types. `RDM` class bounds might be tricky. It's not obvious how a class can be instantiated to `mutable` and `immutable` instances, and the mutability guarantee is not broken. The answer is that constructors in `RDM` classes are

carefully designed so that the same constructor can be reused to create both `mutable` or `immutable` instances.

```
@Immutable
class Car {
  int numberOfWheels = 4;
  ...
}
@Immutable Car imc = new @Immutable Car(); // OK
@Mutable Car mc = new @Mutable Car(); // error.
mc.numberOfWheels = 0; // if mutable instance were allowed, this
    assignment
// would change the abstract state of Car instance
```

```
@ReceiverDependentMutable
class Person {
  ...
}
@Immutable Person imp = new @Immutable Person();// OK
@Mutable Person mp = new @Mutable Person(); // OK
```

java.lang.Object is bounded with @ReceiverDependentMutable, so any valid instantiations of instances are allowed.

4.8.3 Compatability With Super Type Element

The mutability bound of a type declaration should be compatible with the mutability bound of super type declarations traversing all the way up to java.lang.Object. Specifically, a `mutable` class can only have `mutable` subclasses. `immutable` classes can only have `immutable` classes. `RDM` classes can have either `mutable`, `immutable` or `RDM` subclasses. Only the above cases are compatible type declarations.

```
@Immutable
class A {
  int x = 0;
}
@Mutable B extends A {} // error
@Mutable B mb = new @Mutable B();
mb.x++; // breaks the immutability guarantee if viewed as A
```

```
@ReceiverDependentMutable
class A {}
@Mutable
class B extends A {} // OK. Going to mutable instantiations
@Immutable
class C extends A {} // OK. Going to immutable instantiations
@ReceiverDependentMutable
```

```
class D extends A {} // OK. Transfers the receiver dependent
   mutability downward
```

java.lang.Object is bounded with @ReceiverDependentMutable, so any valid bounded class can inherit from java.lang.Object. For example, @Immutable Number `extends` Object, @Mutable AbstractStringBuilder `extends` Object etc.

4.8.4 Fields

All mutability types of fields are allowed except `polymutable`. Two points to mention: since PICO is an abstract immutability type system, `mutable` fields are allowed in `immutable` classes, meaning the field is outside the abstract state of the owner object; `readonly` fields are also allowed in `immutable` classes, even though both `mutable` objects and `immutable` objects may be captured by them, which makes the field out of the abstract state.

We use a slightly different version of the factory design pattern example to illustrate why `polymutable` fields are not allowed:

```
@Immutable
class ProductFactory {

  @PolyMutable Product o = new @PolyMutable Product();

  @PolyMutable Product createProduct() {
    return o;// Return polymutable field directly
  }
}
...
ProductFactory factory = new /*@Immutable*/ ProductFactory();
@Mutable Product mp = factory.createProduct(); // ok
@Immutable Product imp = factory.createProduct(); // ok
```

The single `Product` instance `o` gets both `mutable` and `immutable` aliases. This breaks the soundness of assignment-context sensitivity. Therefore, `polymutable` fields are not allowed.

4.8.5 Field Initializations

There are three ways to initialize fields: on field declarations with field initializers, in initialization blocks or in constructors. Initialization blocks are blocks that initialize fields of a class whenever an instance of that class is created. It can hold the common initialization logic that can be shared by every constructor.

```
class A {
   Object o = new Object(); // On field declarations
   {
     o = new Object(); // In initialization blocks
   }
```

```
  A() {
7    o = new Object(); // In constructors
  }
9 }
```

Every instance field has the receiver object, as this is how instance fields work. When initializing them, accessing and writing to instance fields, their declared types should be viewpoint adapted to the type of their receivers, and the result type should be used as the resolved type of the field.

The receiver type on field declarations and in initialization blocks is the bound of the enclosing class. The receiver type in the constructor has the same mutability type as the constructor return type. For example:

```
1 @Immutable class B {
    // o's type is @Immutable ▷ @ReceiverDependentMutable =
       @Immutable
3   /*@ReceiverDependentMutable*/ Object o = new @Immutable Object
       ();
    {
5     o = new @Immutable Object();// receiver is the bound,
         @Immutable
    }
7   /*@Immutable*/ B() {
      o = new @Immutable Object();// receiver is the constructor
         return, @Immutable
9   }
  }
```

4.8.6 Constructors

The constructor return type determines what kinds of mutability instances can be created. There are three valid constructor returns types: `mutable`, `immutable` and `RDM`. It's obvious that `mutable` constructors can only create `mutable` instances and `immutable` constructors can only create `immutable` instances. `RDA` constructors can create any new instance as long as the `new` expression has a valid type, namely `mutable`, `immutable`, `RDM` or `polymutable`. For example:

```
  @Immutable class A {
2   int x;
    @Immutable A (int x) {
4     this.x = x;
    }
6 }
  @Immutable A ima = new @Immutable A(0); // OK
8 @Mutable A ma = new @Mutable A(2);// error. Constructor
     invocation incompatible
  ma.x = 5;// if ma were valid, x's value would be changed.
     Immutable guarantee is broken.
```

To invoke constructors correctly, the rules are:

- The invoked constructor's return type adapted to the `new` expression's type must be a supertype of the `new` expression's type.
- All arguments must be subtypes of the corresponding result types of viewpoint adapting the invoked constructor's declared parameter types to the `new` expression's type.

The first rule may be surprising at first glance. Because for the assignment of `new` expressions to variables, the type of the `new` instance should be a subtype of the left-hand-side assignment context's type. It may seem possible that the result of adapting the invoked constructor's return to the type of the `new` instance becomes a supertype of the assignment context's type, breaking type soundness. However, PICO's mutability type hierarchy and type rules for valid `new` instance types are designed carefully so that this case can't happen. For `mutable` or `immutable` invoked constructor return types, rule 1 becomes a trivial subtype relation: `new` instance types are the subtypes of the invoked constructor's return type — either `mutable` or `immutable`. So, the soundness-breaking case can't happen. For `RDM` invoked constructor return types, after viewpoint adaptation, rule 1 becomes a trivial subtype relation: the type of `new` expression is a subtype of itself. Again, the possible soundness-breaking case can't happen.

It's not obvious how `RDM` constructors make ensure there aren't soundness-breaking aliases to `RDM` fields (we mean an `RDM` object has both `mutable` and `immutable` references). In brief, they work due to the following two reasons: one is viewpoint adaptation when invoking constructors; the other is that there are no shared `RDM` objects by all instances of a class (because `RDM` is forbidden in static contexts. Section 4.8.15 will discuss why) and one `RDM` object gets attached to one and only one receiver object, so there won't be soundness-breaking aliases to one `RDM` object.

```
1  @ReceiverDependentMutable
   class C {
3    @ReceiverDependentMutable Date d;

5    @ReceiverDependentMutable C(@ReceiverDependentMutable Date d)
        {
       this.d = d;
7    }
   }
9  @Mutable C mc = new @Mutable C(new @Mutable Date());
   mc.d.setMonth(2);// mc.d refers to a mutable Date instance.
      Mutation is allowed
11 @Immutable C imc = new @Immutable C(new @Immutable Date());
   imc.d.setMonth(3);// error. imc.d refers to an immutable Date,
      so mutation is not allowed
```

With viewpoint adaptation, `RDM` fields are resolved to correct mutability types consistently with the receiver, therefore immutability constraint won't break.

A more interesting scenario is creating a `RDM` object inside a `RDM` constructor.

```
@ReceiverDependentMutable
```

```
class D {
  @ReceiverDependentMutable Date d;
  @ReceiverDependentMutable D() {
    d = new @ReceiverDependentMutable Date();
  }
}
@Mutable D md = new @Mutable D();
md.d.setYear(2018);// OK. Newly created RDM Date instance is
   resolved to mutable
@Immutable D imd = new @Immutable D();
imd.d.setYear(2019);// error. Newly created RDM Date instance is
    resolved to immutable
```

`md.d` and `imd.d` refer to two different Date instances on the heap. Since `md` and `imd` are created only once, their mutability information is fixed once created, and the `RDM` instances that they point to get resolved to the same types as the receiver types respectively. By this, there won't be both `mutable` and `immutable` aliases to the same Date instance created at line 5. Therefore, type soundness is still guaranteed.

4.8.7 Cirular Initialization of Immutable Objects

Below is an example (inspired by *Freedom Before Commitment: A Lightweight Type System for Object Initialisation*[23]) to illustrate how circular initialization of `immutable` data structures can be achieved.

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

`this` in the constructor has the underinitialization type in the initialization hierarchy. So passing `this` to the ListNode constructor typechecks (of course mutability types also satisfy subtype relations). `new` ListNode(`this`) returns underinitialization `immutable` ListNode. Since head in the List constructor is unknowninitialization `immutable`, the assignment to `head` typechecks. After the List constructor finishes, both List and ListNode instances finish initialization, and both turn into `immutable` objects, which can't be mutated later on. Mutually dependent `immutable` data structure can thus be easily set up in PICO.

4.8.8 Compatiblity Between Constructor And Type Element Bound

We know that `mutable` classes can only have `mutable` instances, `immutable` classes can only have `immutable` instances, and `RDM` classes can have instances of any mutability. To achieve this restriction, constructor return types should be compatible with the type elements' bounds. The compatability rules between constructor returns and type element bounds are as follows:

- if the type element is `mutable`, the constructor return should be `mutable`
- if the type element is `immutable`, the constructor return should be `immutable`
- if the type element is `RDM`, the constructor return can be `mutable`, `immutable` or `RDM`

Let's see a non-trivial violating example:

```
@Immutable
class A {
  int x = 0;
  @ReceiverDependentMutable A() {} // error. Constructor return
     incompatible
}

@Mutable A ma = new @Mutable A();
ma.x--;
```

If a `RDM` constructor is allowed, then `mutable` instantiation `ma` is allowed. Then, field `x`'s value can be arbitrarily changed. This breaks the immutability guarantee that instances of class A must be `immutable`.

Regarding rule 3, since `RDM` type elements can have instances of any mutability type, having `mutable` and `immutable` constructor return types makes sense — they are only restricting mutability of instances earlier in the constructor level.

4.8.9 Compatability Between Current Constructor and This/Super Constructor

In Java, one constructor can call another constructor declared in the current class (using this()) or super constructor (using super()). For example:

```
class B {
  int x;
  B(int x) {
    this.x = x;
  }
}
class C extends B {
  int y;
  C(int x, int y) {
```

```
    super(x);
    this.y = y;
  }
}
```

When invoking another constructor from one constructor, the rules are:

- The invoked constructor's return type adapted to the invoking constructor's return type must be a supertype of the invoking constructor's return type.
- All arguments must be subtypes of their corresponding result types of viewpoint adapting invoked constructor's declared parameter types to the invoking constructor's return type.

```
@ReceiverDependentMutable
class C {
  Date d;

  @Immutable C() {
    this(new @Immutable Date());// OK
  }

  @ReceiverDependentMutable C(@ReceiverDependentMutable Date d)
     {
    this.d = d;
  }
}
@Immutable C imc = new @Immutable C();
imc.d.setMonth(11);// error. as imc.d refers to immutable Date
   instance.
```

If there is no restriction on invoking other constructors from a constructor, type soundness may be compromised. For example:

```
@ReceiverDependentMutable
class C {
  Date d;

  @Immutable C() {
    this(new @Mutable Date());// error. Invoking mutable
       constructor from immutable constructor
  }

  @Mutable C(@Mutable Date d) {
    this.d = d;
  }
}
@Immutable C imc = new @Immutable C();
```

```
imc.d.setMonth(11);// imc.d is immutable reference, but points
   to mutable Date
```

One thing to note is that in the formalization later, we didn't consider this() constructor invocations because it's not part of the language syntax. But in the implementation, we have the same restrictions on super() and this() constructors invocations.

4.8.10 Compatability Between Type Usage With Type Element Bound

There are rules regarding what kinds of usage types are valid within the type element's bound in terms of mutability. The rules are:

- If the type element is `mutable`, any usage type[5] is allowed except `RDM` or `immutable`
- If the type element is `immutable`, any usage type is allowed except `RDM` or `mutable`
- If the type element is `RDM`, any usage type is allowed

We know that if a type element's bound is `mutable`, PICO only allows `mutable` usages. Forbidding `immutable` usage of a `mutable` type seems straightforward. Forbidding `RDM` is to avoid the possibility of getting `immutable` reference if the receiver object is `immutable`, which breaks type soundness, because this `immutable` reference can never capture a valid instance (because no `immutable` instances exist for `mutable` type elements). However, `polymutable` usages are allowed. The reason behind this is that `polymutable` types are resolved by actual invocations, but actual invocations can only pass `mutable` or `readonly` instances in, making `polymutable` to always be able to resolve to a valid usage type under the corresponding type element's bound. For example:

```
@Mutable
class A {}
class B {
  void foo(@PolyMutable A a){}; // OK.
}
@Mutable B mb = ...;
mb.foo(new @Mutable A());// Clients cannot pass RDM or immutable
    A instances. polymutable is resolved to mutable
```

For `immutable` type elements, similar reasoning applies. For `RDM` type elements, as they are naturally allowed to be instantiated to any valid mutability instances for `new` expressions (`mutable`, `immutable`, `RDM` and `polymutable`), usage types of `RDM` type elements are compatible with all types.

4.8.11 Instance Methods

Instance methods follow the same mutability restrictions on other references and objects. A `readonly` or `immutable` method never modifies the abstract state of the receiver object.

[5] As we said before, `substitutablepolymutable` and `bottom` are always not included

In PICO, the toString(), hashCode(), and equals(Object) methods are specially treated as `readonly` methods: even if there aren't explicit mutability qualifiers on those methods, their declared method receivers are treated as `readonly`. The motivation is that those three methods shouldn't side-effectingly mutate the abstract state of the receiver object.

4.8.12 Instance Methods Invocations

There are no special rules with regards to references inside instance methods. The references inside method bodies follow the same mutability restrictions as other ordinary references do. However, invocations of instance methods should be checked against the viewpoint adapted signature of the method. Viewpoint adaptation happens on the declared method receiver, parameters and the return type. For example:

```
@ReceiverDependentMutable
class A {}
@ReceiverDependentMutable
class Test {
   @ReceiverDependentMutable A a;
   @ReceiverDependentMutable A getA(@ReceiverDependentMutable
      Test this) {
     return a;
   }
}

@Mutable A ma = new @Mutable A();
@Mutable Test mt = new @Mutable Test();
@Mutable A maret = mt.getA(ma);
Assert.equals(ma, maret);

@Immutable A ima = new @Immutable A();
@Immutable Test imt = new @Immutable Test(ima);
@Immutable A imaret = imt.getA();
Assert.equals(ima, imaret);
```

The first invocation of the getA() method on line 13 after viewpoint adaptation becomes:

```
@Mutable A getA(@Mutable Test this);
```

The actual receiver is @Mutable Test, which is a subtype of @Mutable Test after viewpoint adaptation; the viewpoint adapted return type is @Mutable A, which is a subtype of @Mutable A. So the method invocation on line 13 typechecks. The second method invocation on line 18 also typechecks using similar reasoning.

4.8.13 Instance Methods Overriding

Just like `RDA` classes can go to `mutable` or `immutable` (subclasses), instance methods whose declared types contains `RDM` can also become `mutable` or `immutable` in overriding methods. For example:

```java
@ReceiverDependentMutable
class A {
  @ReceiverDependentMutable A identity(@ReceiverDependentMutable
     A this) {
    return this;
  }
}

/*@Mutable*/
class B extends A {
  @Mutable B identity(@Mutable B this) {
    return (B) super.identity(this);
  }
}

@Immutable
class C extends A {
    @Immutabl C identity(@Immutable C this) {
        return (C) super.identity();
    }
}
```

Standard method overriding requires that the overriding method's receiver and parameters are contra-variant to the overriden method, and the return type of the overriding method should be co-variant with the overriden method return. However, this breaks the flexibility of PICO. For example, if a method parameter is `RDM` in the overridden method, an overriding mutable subclass can only override it to `RDM` or `readonly`. We already discussed that `RDM` type usages are not valid for `mutable` type elements. Thus, this `mutable` subclass can only make the parameter type `readonly`. This is too restrictive. The only requirement an overriding method should satisfy is that all existing instance methods should still be callable on subclasses' instances.

PICO checks the validity of overriding method as the following steps:

1. Viewpoint adapt the declared receiver, formal parameters, and the return type of the overridden method to the bound of the type element that encloses the overriding method.
2. Use standard override checks between the overriding method signature and the resulted method signature after viewpoint adaptation.

This overriding rule is called flexible overriding rule in PICO. In the class B, we already know that all instances of B are `mutable`, therefore the method signature is safe to change to the signature: @Mutable identity(@Mutable B this). For the `immutable` subclass C, all instances of it are `immutable`, thus changing the method signature to @Immutable C identity(@Immutable C this) still assures that the method is callable on C's instances.

4.8.14 PolyMutable Methods And Their Resolutions

PICO supports both receiver-context and assignment-context sensitivity features. @PolyMutable receiver adapting @RDM declared type will yield @PolyMutable, which satisfies the condition in which the interactions between the two context sensitivity should be handled carefully. In order to correctly handle the interactions between the two, PICO uses an internal qualifier `substitutablepolymutable` to temporarily represent what `polymutable` used to represent: once the method with at least one `polymutable` is invoked, PICO replaces every occurrence of `polymutable` with `substitutablepolymutable`, then performs the viewpoint adaptation of the method signature. For example:

```
@ReceiverDependentMutable
class C {
  @PolyMutable List<@ReceiverDependentMutable Date>
     getDateWrapped
    @Readonly C this, @ReceiverDependentMutable Date date) {
       List<@ReceiverDependentMutable Date> l = new ArrayList
          <>();
       l.add(date);
       return new @PolyMutable ArrayList<>(l);
  }
}

@Immutable C imc = new @Immutable C();
@Mutable List<@Immutable Date> ml = imc.getDateWrapped(new
   @Immutable Date());
@Immutable List<@Immutable Date> iml = imc.getDateWrapped(new
   @Immutable Date());
```

The above two method invocations are checked in the below way:

1. Replace all @PolyMutable occurrences with @SubstitutablePolyMutable. The method signature becomes:

   ```
   @SubstitutablePolyMutable List<@ReceiverDependentMutable
      Date> getDateWrapped(
     @Readonly C this, @ReceiverDependentMutable Date date)
   ```

2. Perform viewpoint adaptation. Because the receiver object `imc` is `immutable`, the signature becomes:

   ```
   @SubstitutablePolyMutable List<@Immutable Date>
      getDateWrapped(
     @Readonly C this, @Immutable Date date)
   ```

3. Resolve the solution for `substitutablepolymutable`:

   - In the first invocation on line 12, generate constraint:

```
@SubstitutablePolyMutable <: @Mutable // resolved to
   @Mutable
```

Therefore the method signature becomes:

```
1 @Mutable List<@Immutable Date> getDateWrapped(@Readonly C
     this,
  @Immutable Date date)
```

The actual receiver is a subtype of the result receiver type; arguments are subtypes of result parameter types; the result return type is a subtype of the left-hand-side type, so the method invocation typechecks.

- In the second invocation on line 13, generate constraint:

```
@SubstitutablePolyMutable <: @Immutable // resolved to
   @Immutable
```

Therefore the method signature becomes:

```
1 @Immutable List<@Immutable Date> getDateWrapped(@Readonly
     C this,
  @Immutable Date date)
```

Similar checks as above indicate the second method invocation also typechecks.

If `polymutable` wasn't replaced to another internal qualifier before viewpoint adaptation, there might be a way of getting `polymutable` on the method signature even though it's not declared on the method, but instead the result of viewpoint adaptation. To illustrate this:

```
  class C {
2   void foo(@ReceiverDependentMutable Date date);
  }
4 @PolyMutable C c;
  c.foo(new @Mutable Date()); // error. Should pass polymutable
     argument
```

After invoking the foo() method, there would be a `polymutable` on the result type of parameter `date` as the result of viewpoint adaptation. This `polymutable` shouldn't be resolved, because the foo() method wasn't declared to be assignment-context sensitive, but instead should be receiver-context sensitive. So, in order to differentiate such a case, replacement of `polymutable` is always performed when a method with at least one `polymutable` is invoked before viewpoint adaptation happens, and the new type is `substitutablepolymutable`, an internal qualifier that can never be written by users. Not being able to explicitly write `substitutablepolymutable` avoids the case that `substitutablepolymutable` appears on method signatures not by replacing `polymutable`.

For static methods that don't have viewpoint adaptation, the `polymutable` replacement still happens when they are invoked.

PICO forbids @PolyMutable on constructor parameters. The reason is that @PolyMutable fields are not allowed, therefore, it's only possible to capture @PolyMutable constructor arguments in @Readonly fields. However, in this case, declaring the constructor parameters to @Readonly is simpler.

4.8.15 Static Context

The static context of a class includes static fields, static blocks and static methods. They are not bound to a particular instance. Instead, they are shared by all the instances of the class. So naturally, they don't have receivers to access them. It's a violation of the well-formedness constraint if @ReceiverDependentMutable is used on static fields, static blocks or on static methods.

One violating example that illustrates this:

```
@ReceiverDependentMutable
class A {
  static @ReceiverDependentMutable Date sd = new
     @ReceiverDependentMutable Date();
  @ReceiverDependentMutable A() {}
}
@Mutable A ma = new @Mutable A();
ma.sd.setYear(2019); // Allowed, since ma is mutable, ma.sd
   become mutable

@Immutable A ima = new @Immutable A();
ima.sd.setYear(2020); // Error, ima is immutable, ima.sd is
   immutable
```

If @ReceiverDependentMutable is allowed on static field `sd`, the single Date instance in static field sd gets `mutable` and `immutable` references to it. The mutability of that instance becomes unclear, which breaks the soundness of PICO.

`RDA` is also fobidden due to the same reason on static fields. Static fields are either `final` or `assignable`.

4.8.16 Possible Loophole Of Assignable Fields

There is one case in which `assignable` field is forbidden to be re-assigned.

- If the receiver is `readonly`, and the field is `assignable RDM`, this assignment is forbidden.

If this is not the case, there will be a loophole in PICO to compromise `immutable` object's abstract state.

```
@Immutable
class C {
  @Assignable /*@ReceiverDependentMutable*/ Date date;
  @Immutable C(@Immutable Date date) {
    this.date = date;
  }
}

@Immutable Date imdate = new @Immutable Date();
```

```
@Immutable C imc = new @Immutable C(imdate);
@Readonly C roc = imc;
@Mutable Date mdate = new @Mutable Date();
roc.date = mdate; // If this assignment were OK, roc.date would
    capture mutable Date instance. However notice that roc is a
    readonly reference to immutable C, therefore imc.date is
    actually immutable reference but points to mutable Date object
     now
```

It looks like the assignment `roc.date = mdate` is OK according to the discussions before: `date` is assignable, so re-assignment is allowed; `roc.date` is readonly, which is a supertype of `mdate`. However, this will cause `immutable` reference `imc.date` to point to a `mutable` Date instance, which breaks the type soundness. Therefore, we forbid field assignment in this specific case to ensure type soundness.

This problem is quite analogous to the writable fields' variance issues. Subclasses can't use co- or contra-variance for fields. In PICO, writes to fields are not allowed through `readonly` references. Therefore, it's OK to use co-variant fields types for `RDM` fields from `readonly` receivers (equivalent to superclasses) to `mutable` or `immutable` receivers (equivalent to subclasses). However, this readonly restriction is broken by `assignable` fields: they can be re-assigned now. We have to forbid this combination in order to achieve type safety.

4.8.17 Arrays

Arrays are special construct different from other reference types. There is no declaration of arrays as other type elements do. However, array write and array read still have receiver, so their mutability is also enforced by PICO. For example,

```
class C {
  Date /*@ReceiverDependentMutable*/ [] f = new Date /*@Mutable
     */ [3];// f is declared RDM array of mutable Date
  void foo() {
    f[0] = new Date();// OK. f[0] has mutable receiver.
       Assignment is allowed
  }
  void bar(@Readonly C this) {
    f[0] = new Date();// error. f[0] has readonly receiver,
       assignment is forbidden.
  }
}
```

Java allows co-variant subtyping for array components. For example:

```
Object[] o = new String [2];// Allowed in Java
```

Java allows a String array to be assigned to an Object array. When writing to `o`, Java has runtime checks to ensure that the value is of the String type, to make sure only Strings are stored in Object array `o`.

PICO adopts the same co-variant array component subtype relations as if there are additional runtime checks in terms of mutability and writes to arrays can be performed safely, even though there are no runtime components for PICO. We found in practice, doing so is more flexible compared to only allowing invariant array component types, and it's also consistent with Java's implementation.

This co-variant subtyping for array components is actually the default behavior of Checker Framework at the moment. There is an option "-AinvariantArrays" to enable invariant array component subtyping checks to address the lack of runtime enforcement. In this mode, array components should be the same type in subtype relations.

4.8.18 Type Casts

Casting is a mechanism in Java to circumvent the compiler's static type checking and delegate the safety check to runtime. With casts, an arbitrary conversion of types is allowed by the compiler, even though the conversion might not be safe at runtime.

Because PICO is a static type system, what it can do at most is to warn about possible unsafe casts, if they are not always-safe upcasts. It's only a warning, because the cast might be safe. It would be annoying if PICO always reported errors for them. PICO only issues warning, and brings the possible unsafe casts to programmers' attention.

Casts sometimes can be useful to initialize a complex data structure as `mutable`, and then cast it to `immutable` if there are no aliases to the `mutable` data structure. For example:

```
1 class C {
    @Immutable Graph buildGrapgh() {
3     @Mutable Graph g = new @Mutable Graph();
      ...complex initialization to g...
5     return (@Immutable Graph) g;// no alises to g exist after
         method returns
    }
7 }
```

The cast to `immutable` type is actually safe, because there are no aliases that may side-effect the returned graph. In such cases, provided that programmers make sure no aliases exist for the `mutable` data structure, the `mutable` data structure can be cast to `immutable`, which is normally not compatible with `mutable`[6]. This is another motivation for not reporting incompatible cast as errors.

4.9 Formalization

We formalized PICO for a minimum core set of language constructs that PICO supports. Java's generics are not reflected in the formalization, but in the implementation, similar to GUT, generics are also supported. In future work, we'll formalize generics.

[6] "Incompatible" means neither type is a subtype of the other, between two types

4.9.1 Language Syntax Definition

We use A-normal form (ANF) to define the syntax of the language[1]. In ANF, instead of passing expressions as arguments to functions, local variables are used to store the expressions first, and then those local variables are passed as trivial arguments to functions.

$$P ::= \overline{cd} \qquad \textit{(program)}$$

$$cd ::= q_C \ \texttt{class} \ C \ \texttt{extends} \ D \ \{\overline{fd} \ kd \ \overline{md}\} \qquad \textit{(class)}$$

$$fd ::= a \ q \ C \ f \qquad \textit{(field)}$$

$$kd ::= q \ C \ (\overline{t \ C \ g}, \overline{t \ C \ f}) \ \{ \ \texttt{super}(\overline{g}); \ \texttt{this}.\overline{f} \ = \ \overline{f}; \} \qquad \textit{(constructor)}$$

$$md ::= t \ C \ m \ (t \ C \ \texttt{this}, \overline{t \ C \ x}) \ \{ \overline{t \ C \ y \ s}; \ \texttt{return} \ z; \} \qquad \textit{(method)}$$

$$e ::= x \mid x.f \qquad \textit{(expression)}$$

$$s ::= x = e \mid x.f = y \mid x = y.[q]m(\overline{z}) \mid x = \texttt{new} \ [k] \ q \ C(\overline{y}) \mid s; s \qquad \textit{(statement)}$$

$$t ::= k \ q \qquad \textit{(qualifier type)}$$

$$k ::= \texttt{initialized} \mid \texttt{underinitialization} \mid \texttt{unknowninitialization} \mid \texttt{fbcbottom} \qquad \textit{(initialization qualifier)}$$

$$q ::= \texttt{readonly} \mid \texttt{mutable} \mid \texttt{polymutable} \mid \texttt{substitutablepolymutable} \mid \texttt{receiverdependentmutable} \mid \texttt{immutable} \mid \texttt{bottom} \qquad \textit{(mutability qualifier)}$$

$$a ::= \texttt{assignable} \mid \texttt{receiverdependentassignable} \mid \texttt{final} \qquad \textit{(assignability qualifier)}$$

Figure 4.3: Syntax of language

- Among mutability qualifiers, `substitutablepolymutable` and `bottom` are not part of the surface syntax — they are only internal qualifiers that can't be written by users.
- In method invocations, [q] is the substitution for `substitutablepolymutable`, but users cannot write it explicitly. Inference of q is already discussed in section 3.3. Here it is made explicit for a better explanation and easier formalization.
- In new instance creations, [k] is not allowed to be explicitly written according to FBC type rules. Instead, it is inferred by the type rules of FBC (see section 2.4 for more details).
- For simplicity, program P is always available in helper functions and type judgements.

4.9.2 Type Environment

$\Gamma = (x_1 : k_1 \ q_1 \ C_1, x_2 : k_2 \ q_2 \ C_2, ..., x_n : k_n \ q_n \ C_n)$

$\Gamma(x) = k \ q \ C$

$\Gamma(x) = k \ q$ is a shorthand for $\Gamma(x) = k \ q \ _$

4.9.3 Subtype Relations

$k_1\ q_1 <: k_2\ q_2 \iff k_1 <: k_2 \wedge q_1 <: q_2$

4.9.4 Helper Function

- fType(C, f) = a q
- bound(C) = q_C, in which C is a class
- constructor(C) = kd, in which C is a class
- typeof(C, m) = $(k_{\text{this}}\ q_{\text{this}}, \overline{k_p\ q_p} \rightarrow k_{\text{ret}}\ q_{\text{ret}})$
- replacedtypeof(C, m) = $(k_{\text{this}}\ q'_{\text{this}}, \overline{k_p\ q'_p} \rightarrow k_{\text{ret}}\ q'_{\text{ret}})$ in which:
  $q'_{\text{this}} = q_{\text{this}}$ [`substitutablepolymutable/polymutable`],
  $\overline{q'_p} = \overline{q_p}$ [`substitutablepolymutable/polymutable`],
  $q'_{\text{ret}} = q_{\text{ret}}$ [`substitutablepolymutable/polymutable`],
  typeof(C, m) = $(_\ q_{\text{this}}, \overline{_\ q_p} \rightarrow _\ q_{\text{ret}})$

4.9.5 Viewpoint Adaptation Rules

q $\triangleright$ receiverdependentmutable = q
$_ \triangleright$ q = q (otherwise)

4.9.6 Typing Rules

4.9.6.1 Expressions

$$\text{T-VAR}\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{T-FLD}\ \frac{\begin{array}{c}\Gamma(x) = k_x\ q_x\ C \quad fType(C, f) = _\ q_f \quad q = q_x \triangleright q_f \\ k_x = initialized \Rightarrow k = initialized \\ k_x \neq initialized \Rightarrow k = unknowninitialization\end{array}}{\Gamma \vdash x.f : k\ q}$$

4.9.6.2 Statements

$$\text{T-VARASS}\ \frac{\Gamma \vdash e : t_e \quad t_e <: \Gamma(x)}{\Gamma \vdash x = e}$$

$$\text{T-FLDASS}\ \frac{\begin{array}{c}\Gamma(x) = k_x\ q_x\ C \quad \Gamma(y) = k_y\ q_y \quad fType(C, f) = a_f\ q_f \\ q_x = mutable \\ \vee\ (k_x = underinitialization \wedge q_x = immutable) \\ \vee\ (k_x = underinitialization \wedge q_x = receiverdependentmutable) \\ \vee\ (a_f = assignable \wedge (q_x \neq readonly \vee\ q_f \neq receiverdependentmutable)) \\ q_y <: q_x \triangleright q_f \quad k_x = underinitialization \vee k_y = initialized\end{array}}{\Gamma \vdash x.f = y}$$

- Every assignment to instance fields without an explicit receiver has implicit receiver `this`. In constructors, $q_{\text{this}} = q_{\text{ret}}$. In initialization blocks, $q_{\text{this}} = \mathit{bound}(C)$ where C is the class that encloses the initialization block. In instance field declarations (with initializers), $q_{\text{this}} = \mathit{bound}(C)$ where C is the class that encloses fields f.
- Final fields are enforced by the Java compiler and doesn't need PICO to do anything.

$$\text{T-CALL}\ \frac{\begin{array}{c}\Gamma(x) = k_x\ q_x \quad \Gamma(y) = k_y\ q_y\ C \quad \Gamma(\overline{z}) = \overline{k_z\ q_z} \\ \mathit{replacedtypeof}(C, m) = k_{\text{this}}\ q'_{\text{this}}, \overline{k_p\ q'_p} \rightarrow k_{\text{ret}}\ q'_{\text{ret}} \\ k_y <: k_{\text{this}} \quad \overline{k_z <: k_p} \quad k_{\text{ret}} <: k_x \\ q_{\text{this-vp}} = q_y \triangleright q'_{\text{this}} \quad \overline{q_{\text{p-vp}}} = q_y \triangleright \overline{q'_{\text{p}}} \quad q_{\text{ret-vp}} = q_y \triangleright q'_{\text{ret}} \\ q_{\text{this-vp}} = \mathit{substitutablepolymutable} \Rightarrow q_y <: q \\ q_{\text{this-vp}} \neq \mathit{substitutablepolymutable} \Rightarrow q_y <: q_{\text{this-vp}} \\ \overline{q_{\text{p-vp}}} = \mathit{substitutablepolymutable} \Rightarrow \overline{q_z} <: q \\ \overline{q_{\text{p-vp}}} \neq \mathit{substitutablepolymutable} \Rightarrow \overline{q_z} <: \overline{q_{\text{p-vp}}} \\ q_{\text{ret-vp}} = \mathit{substitutablepolymutable} \Rightarrow q <: q_x \\ q_{\text{ret-vp}} \neq \mathit{substitutablepolymutable} \Rightarrow q_{\text{ret-vp}} <: q_x \end{array}}{\Gamma \vdash x = y.[q]m(\overline{z})}$$

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

$$\text{T-NEW}\ \frac{\begin{array}{c}\Gamma(x) = k_x\ q_x \quad \Gamma(\overline{y}) = \overline{k_y\ q_y} \quad \mathit{typeof}(C) = \overline{k_{\text{p}}\ q_{\text{p}}} \rightarrow q_{\text{ret}} \\ \overline{q_{\text{y}}} <: q \triangleright \overline{q_{\text{p}}} \quad q <: q \triangleright q_{\text{ret}} \quad q \neq \mathit{readonly} \quad \overline{k_y <: k_p} \quad q <: q_x \quad k <: k_x \\ \overline{k_p} = \mathit{initialized} \Rightarrow k = \mathit{initialized} \quad \overline{k_p} \neq \mathit{initialized} \Rightarrow k = \mathit{underinitialization} \end{array}}{\Gamma \vdash x = \texttt{new}\ [k]\ q\ C(\overline{y})}$$

$$\text{T-SEQ}\ \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2}$$

4.9.7 Well-formedness Rules

$$\text{WF-FLD}\ \frac{\mathit{fType}(fd) = _\ q \quad q \neq \mathit{polymutable} \quad C <: D}{\vdash_C \mathit{fd\ is\ OK}}$$

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

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

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

$$\text{WF-CLASS}\ \frac{\begin{array}{c}\vdash_C \overline{fd}\ is\ OK \quad \vdash_C kd\ is\ OK \quad \vdash_C \overline{md}\ is\ OK \\ C <: D \quad \vdash D\ is\ OK \quad \vdash C <: D\ is\ OK \\ q_{\text{C}} = mutable \vee q_{\text{C}} = immutable \vee q_{\text{C}} = receiverdependentmutable\end{array}}{\vdash q_C\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}; kd\ \overline{md}\}\ is\ OK}$$

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

4.9.8 Extension to Real Java With Static And Blocks

In real Java, there are static fields, static methods and initialization blocks.

4.9.8.1 Helper Method

usedQualifiers($\overline{s}$) returns all mutability qualifiers used in $\overline{s}$ recursively.

4.9.8.2 Revised And Extended Language Syntax

$$\begin{array}{rll} cd & ::= q_C\ \texttt{class}\ C\ \texttt{extends}\ D\{\overline{sfd}\ \overline{fd}\ \overline{sib}\ \overline{ib}\ kd\ \overline{smd}\ \overline{md}\} & \textit{(class)} \\ sfd & ::= \texttt{static}\ q\ a\ C\ f & \textit{(static field)} \\ smd & ::= \texttt{static}\ t\ C\ sm\ (\ t\ C\ x)\{\overline{t\ C\ y\ s};\ \texttt{return}\ z;\} & \textit{(static method)} \\ sib & ::= \texttt{static}\ \overline{s}; & \textit{(static initialization block)} \\ ib & ::= \overline{s}; & \textit{(initialization block)} \end{array}$$

Figure 4.4: Revised and extended syntax of language

4.9.8.3 Revised And Extended Well-formedness Rules

$$\text{WF-STATIC-FLD}\ \frac{\begin{array}{c}fType(sfd) = a\ q \quad q \neq polymutable \quad q \neq receiverdependentmutable \\ a \neq receiverdependentassignable\end{array}}{\vdash_C sfd\ is\ OK}$$

$$\text{WF-STATIC-METH}\frac{\begin{array}{c}\vdash (\overline{p} : \overline{k_p\ q_p}, \overline{y} : \overline{k_{\text{local}}\ q_{\text{local}}}) \quad \vdash \overline{s} \quad \vdash \Gamma(z) <: t_{\text{ret}} \\ \overline{q_p} \neq \mathit{receiverdependentmutable} \quad q_{\text{ret}} \neq \mathit{receiverdependentmutable} \\ \mathit{receiverdependentmutable} \notin \mathit{usedQualifiers}(\overline{s}; \mathit{returnz})\end{array}}{\vdash_C \texttt{static}\ t_{\text{ret}}\ C\ sm\ (\ t_{\text{p}}\ C\ p)\{\overline{\ t\ C\ y\ s};\ \texttt{return}\ z;\}\ \mathit{is\ OK}}$$

$$\text{WF-STATIC-BLK}\frac{\Gamma \vdash \overline{s} \quad \mathit{receiverdependentmutable} \notin \mathit{usedQualifiers}(\overline{s})}{\vdash_C \texttt{static}\ \overline{s}\ \mathit{is\ OK}}$$

$$\text{WF-BLK}\frac{\Gamma \vdash \overline{s}}{\vdash_C \overline{s}\ \mathit{is\ OK}}$$

$$\text{WF-CLASS}\frac{\begin{array}{c}\vdash_C \overline{sfd}\ \mathit{is\ OK} \quad \vdash_C \overline{fd}\ \mathit{is\ OK} \quad \vdash_C \overline{sib}\ \mathit{is\ OK} \quad \vdash_C \overline{ib}\ \mathit{is\ OK} \quad \vdash_C kd\ \mathit{is\ OK} \\ \vdash_C \overline{smd}\ \mathit{is\ OK} \quad \vdash_C \overline{md}\ \mathit{is\ OK} \quad C <: D \quad \vdash D\ \mathit{is\ OK} \quad \vdash C <: D\ \mathit{is\ OK} \\ q_{\text{C}} = \mathit{mutable} \vee q_{\text{C}} = \mathit{immutable} \vee q_{\text{C}} = \mathit{receiverdependentmutable}\end{array}}{\vdash q_C\ \texttt{class}\ C\ \texttt{extends}\ D\{\overline{sfd}\ \overline{fd}; \overline{sib}\ \overline{ib}\ kd\ \overline{smd}\ \overline{md}\}\ \mathit{is\ OK}}$$

Chapter 5

Implementation And Experiments — PICO

This chapter discusses the implementation of the PICO type system and PICOInfer, the inference system for PICO, and experiments on real-world open-source libraries. Section 5.1 gives implementation-level information about PICO type checker and PICOInfer. Section 5.2 shows experiment results of PICO type checker and PICOInfer on real-world projects.

5.1 Implementation

PICO type checker and PICOInfer are implemented in the same project called immutability. Unfortunately, we had to implement PICO type checker and PICOInfer separately because inference related classes such as InferenceVisitor, only support one hierarchy in the Checker Framework Inference, but PICO type checker needs support two hierarchies.

The core logic for PICO type checker is composed of 12 files and 1481 lines of non-comment, non-blank[1] Java code. The test cases for PICO type checker consist of 92 files and 2150 lines of Java code.

The core logic for PICOInfer is composed of 10 files and 1469 liens of Java code. The test cases for PICOInfer consist of 52 files and 576 lines of Java code.

In total, immutability project is composed of 178 files, 1298 lines of comments and 5788 lines of Java code. We used local version of upstream dependency Checker Framework and Checker Framework Inference to facilitate the development of PICO.

5.1.1 PICO Type Checker

PICO type checker is developed on Checker Framework. It is a type checking checker that enforces the full features of PICO (initialization, mutability and assignability).

There is another annotation @AbstractSateOnly that's only used on method declarations to restrict a method body to only depend on fields that are in the abstract state of the enclosing

[1]Won't be mentioned again. By Java code, we mean non-comment and non-blank lines

class. For method invocations inside @AbstractSateOnly methods, those methods should also be @AbstractSateOnly methods recursively. If violations happen, PICO type checker will issue a warning for it.

PICO internally treats the hashCode(), equals(Object) methods to be @AbstractSateOnly without even explicit usage of @AbstractSateOnly. The benefit is that the @AbstractSateOnly behavior is predictable and can be determined completely by the abstract state of the receiver object. For example, an `immutable` object's hashCode() method is guaranteed to return the same result if hashCode() depends solely on the abstract state, because the abstract state of `immutable` objects is protected by PICO.

We didn't make @AbstractStateOnly part of PICO's core language, because it's only an application of PICO in experiments. There are still some problems regarding it. First, it conservatively warns about every instance method invocation from @AbstractStateOnly methods, if the invoked method isn't annotated with @AbstractStateOnly. But the result of the invoked method may not flow into the returned result of @AbstractStateOnly methods. For example, System.out.println(). Second, it doesn't support internal cache fields that cache expensive computations' results. Those cache fields are `assignable`, therefore they are outside the abstract state. Accessing them from @AbstractStateOnly methods such as hashCode() is actually safe. However, they will still be warned. So we only mentioned @AbstractStateOnly in the implementation chapter.

5.1.2 PICOInfer

PICOInfer is the inference system built on top of Checker Framework Inference. PICOInfer only supports inferring mutability qualifiers. @PolyMutable is not supported because inference of it is tricky: we need to track all the usages of an assignment-context sensitive method, and find a solution that satisfies all those invocations at each assignment context. This is rather complicated so we decided to not infer @PolyMutable.

Due to the limitations of Checker Framework Inference, that only one type hierarchy can be inferred at a time, initialization qualifiers can't be inferred. There is also no inference system for FBC type system, so that we can infer in two passes, one for initialization qualifiers and the other for mutability qualifiers. Therefore, for initialization hierarchy, PICOInfer assumes @Initialized default for almost every type usage. However, PICOInfer does recognize any existing initialization qualifier, @UnderInitialization, in the source code, and generates the corresponding set of constraints, so that the interaction between initialization and mutability types is consistent with PICO type rules. Users can manually annotate source programs with @UnderInitialization such as on helper methods that are only called from constructors. For constructors and initialization blocks in which receivers are @UnderInitialization but there doesn't exist an explicit tree[2] to be annotated with @UnderInitialization, PICOInfer is developed to be able to handle these cases so that it also generates correct constraints for these language constructs. Other initialization qualifiers don't affect how constraints are generated on mutability side, therefore PICOInfer doesn't need to be aware of their existence.

Assignability qualifiers are also not inferrable, as they are not even part of types. The default @ReceiverDependentAssignable for instance fields is assumed. However, if there are existing

[2]See https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/Tree.html

@Assignable annotations in the source code, PICOInfer is also aware of those, and relaxes the restriction that the receiver (if there is one) must be `mutable`.

Generally, PICOInfer is guaranteed to infer correct mutability qualifiers if the input program typechecks in FBC type system and assignability qualifiers are correctly used.

5.2 Experiments

We ran PICO type checker and PICO inference on 14 real world open-source projects to evaluate whether PICO raises reasonable errors and PICOInfer gives valid inference solutions. Figure 5.1 shows the size of those 14 projects in different dimensions.

Project Name	File	Blank	Comment	Sloc
bdoorProto	8	52	81	152
AFHbackdoor	5	49	26	161
FaceDetection	14	162	72	538
imgscalr	11	302	2186	1181
ECC-RSA-Backdoor	5	416	332	1251
jump	23	770	3211	1937
jdepend	22	869	942	2557
jama	10	589	931	2599
exp4j	30	819	761	5129
jdeb	73	1471	1919	5199
jReactPhysics3D	84	3250	5549	9468
react	63	1147	6338	10095
JLargeArrays	17	882	2496	11337
jblas	72	3341	6738	11821

Figure 5.1: Benchmark projects for running PICO and PICOInfer

Brief introductions of the projects:

- **bdoorProto** backdoor network server/client for debugging/developing running apps
- **AFHbackdoor** Arduino Hacking Framework incubating backdoor
- **FaceDetection** Face Detection app based on Viola-Jones algorithm
- **imgscalr** simple Java image-scaling library implementing Chris Campbell's incremental scaling algorithm as well as Java2D's "best-practices" image-scaling techniques.
- **ECC-RSA-Backdoor** A backdoor based on Error correcting codes
- **jump** Java-based extensible high-precision math package
- **jdepend** a Java package dependency analyzer that generates design quality metrics
- **jama** a basic linear algebra package for Java

- **exp4j** A tiny math expression evaluator for the Java programming language
- **jdeb** This library provides an Ant task and a Maven plugin to create Debian packages from Java builds in a truly cross platform manner.
- **jReactPhysics3D** 3D physics engine written in Java. Port of the ReactPhysics3D C++ physics library.
- **react** Real-time 3D physics library for Java, based on the ReactPhysics3D C++ library
- **JLargeArrays** library of one-dimensional arrays that can store up to 2ˆ63 element
- **jblas** A matrix library for Java which uses existing high performance BLAS and LAPACK libraries like ATLAS.

We first ran PICO type checker on them, and got the number of errors on each project. We manually examined the error results and analyzed if they make sense or not. We iteratively changed implicit and default qualifiers several times, so that the error messages for unannotated programs reduced to a reasonable number. We also got the conclusion that for unannotated programs, it is impossible to be 100% free of errors that might be not real bugs. There are several ground truths that we believe make sense, and we propagate them in unannotated programs. For example, the toString() method should be @Readonly. We'll discuss unavoidable errors in section 5.2.1. However, on the test cases that violate mutability guarantees, PICO type checker nevers gives false positive errors.

We then ran PICOInfer on the same set of projects, and all of the projects successfully get solutions, with partial manual initialization/assignability qualifiers or no manual annotations at all. But the tool we used to insert solutions back to the source code, annotation-tools, throws exceptions and fails to insert 9 projects' solutions back to the source code due to its bugs. We had a workaround for the bug, and we were able to insert solutions to the source code, but we found in one project (maybe there are more), some solutions were inserted to wrong locations, so we gave up the workaround; due to Checker Framework Inference bugs, some valid source code locations get inferred solutions, but those solutions were not written to the jaif file, a file that is the input to annotation-tools to insert solutions; even though some solutions exist in the jaif file, annotation-tools wasn't able to insert them. Therefore, we were not able to typecheck the inferred solutions and verify the correctness of the solutions. But for projects that were successfully inserted solutions to the correct locations, they are mostly annotated, thus becoming easier for humans to continue filling in the missing annotations. We manually examined part of the inserted source code of projects on which the inference and insertion both succeeded[3], and checked whether each PICO type rule is correctly reflected in the result. After the examination, we conclude that the inference results we examined do make sense and obey the full PICO type rules. We also manually filled in missing annotations to source code after inference for the project jama by manually inferring what annotations PICOInfer may have inferred. In the end, we ran PICO type checker and verified that the inferred and manually inserted solutions together completely typecheck.

For PICOInfer, we evaluated its performance, such as running time, constraint sizes, slot sizes, Sat4J[12] solver related sizes: CNF (Conjunctive Normal Form) variable sizes, and CNF clause sizes etc.

[3]The problems mentioned above still exist: some source code locations still miss annotations

5.2.1 PICO Type Checker

Project Name	Initialization Errors	Mutability Errors	Warnings
bdoorProto	0	0	0
AFHbackdoor	0	0	0
FaceDetection	0	6	0
imgscalr	0	44	0
ECC-RSA-Backdoor	3	2	0
jump	1	12	1
jdepend	35	29	15
jama	4	0	0
exp4j	0	0	0
jdeb	3	31	0
jReactPhysics3D	26	13	34
react	12	12	28
JLargeArrays	0	0	0
jblas	12	30	8

Figure 5.2: Categorization of errors by PICO type checker on unannotated benchmarks

5.2.1.1 Initialization Errors

Calling instance methods from constructors

Most of the initialization errors are raised because of calling instance methods within constructors according to experiment results. These invocations from constructors are potential problems when subclasses override the instance method and access subclass fields inside the instance methods, because subclass fields haven't been initialized yet when the super constructor is being executed. This is due to Java's restriction of initialization: the super constructor must be called first, before subclasses's one get called. As a result, NullPointerException might be thrown. FBC type system captures this type of errors and they really reflect flaws in programs.

```
// react/src/main/java/com/flowpowered/react/math/Vector2.java
   :68:
error: [method.invocation.invalid] call to setAllValues(float,
   float) not allowed on the given receiver.
        setAllValues(x, y);
                   ^
  found   : @UnderInitialization(java.lang.Object.class)
     @Mutable Vector2
  required: @Initialized @Mutable Vector2
```

Passing `this` as arguments to new instance creations

Another common initialization error is passing constructor implicit receiver `this` as an argument to `new` instance creation expressions. Since the corresponding constructor parameter is unannotated, it's defaulted to @Initialized, therefore passing `this` (whose type is @UnderInitialization) caused `argument.not.compatible` errors.

```
// react/src/main/java/com/flowpowered/react/collision/
   CollisionDetection.java:72:
error: [argument.type.incompatible] incompatible types in
   argument.
        mBroadPhaseAlgorithm = new SweepAndPruneAlgorithm(this);
                                                          ^
  found   : @UnderInitialization(java.lang.Object.class)
     @Mutable CollisionDetection
  required: @Initialized @Mutable CollisionDetection
```

5.2.1.2 Mutability Errors

Passing implicitly `immutable` types to defaulted `mutable` parameter

Method parameters in completely unannotated programs are defaulted to @Mutable (except those implicitly `immutable` types). For example, if a method parameter is of Object type, then it's equivalent to having @Mutable Object. If, for example, an integer value is passed to the method, then there is an error reported saying `immutable` types cannot be assigned to `mutable` types. One example:

```
// imgscalr/src/main/java/org/imgscalr/Scalr.java:658
error: [argument.type.incompatible] incompatible types in
   argument.
            log(0, "Applying %d BufferedImageOps...", ops.length
               );
                                                         ^
  found   : @Initialized @Immutable int
  required: @Initialized @Mutable Object
```

The signature of the log() method:

```
static void log(int depth, String message, /*Mutable*/ Object...
    params);
```

Only when the log() method ensures that `params` are only read and has @Readonly Object array component type for `params`, PICO can pass `immutable` values safely to the log method.

An unannotated `CharSequence` parameter is another example. Passing `immutable` String would cause an argument incompatible error. The reason why CharSequence is `RDM` is that it has the `mutable` StringBuilder subclass, because StringBuilder is intended for appending strings. So, CharSequence cannot be `immutable`, otherwise it'll break compatability between type element with its super bound. As we discussed before, `RDM` type element usages are defaulted to `mutable` in every location except instance fields. Therefore, passing `immutable` String to unannotated CharSequence also raises errors.

Invoking non-`readonly` methods from `readonly` methods

For completely unannotated programs, the only source of `readonly` methods are the toString(), hashCode() and equals(Object). If they again call other helper methods in the body, since the invoked method is unannotated, we can only assume they are `mutable` and those methods cannot be invoked from a `readonly` context. Therefore, errors are raised. One example:

```
// ECC-RSA-Backdoor/src/rsa/BinNum.java:897
error: [method.invocation.invalid] call to length() not allowed
   on the given receiver.
    @Override
    public String toString(){
        ...
        for(int i=this.length()-1;i>=0;i--){
                                ^
        ...
    }
  found   : @Initialized @Readonly BinNum
  required: @Initialized @Mutable BinNum
```

```
    public int length(/*@Mutable*/ BinNum this){
        return num.size();
    }
```

The implementation of the length() method is actually `readonly`, but PICO, as a type system, doesn't have the ability to infer this knowledge, except there is an explicit @Readonly annotation on the declared receiver of the length() method. Then, the method invocation would typecheck. The length() method might be trivial to annotate with `readonly`, however, for more complex methods whose names are not trivial, annotating with @Readonly does benefit programmers to better understand their code and achieve mutation-free safety.

Passing defaulted `mutable` types as map keys

Map keys expect `immutable` objects in PICO. Unannotated types are by default `mutable`, therefore there would be type argument type incompatible errors if they are used as map keys. For example:

```
// jReactPhysics3D/src/main/java/net/smert/jreactphysics3d/
   collision/broadphase/SweepAndPruneAlgorithm.java:66
error: [type.argument.type.incompatible] incompatible types in
   type argument.
    protected final Map</*Mutable*/ CollisionBody, Integer>
       mapBodyToBoxIndex;
                        ^
  found   : @Initialized @Mutable CollisionBody
  required: @Initialized @Immutable Object
```

Custom subclasses of java.lang.Number are not by default @Immutable

Except the boxed primitive types and several listed subclasses of `Number` listed in section 4.6, custom subclasses of (java.lang.)Number get defaulted to @Mutable, which is not compatible with the super class Number's bound, @Immutable, thus there are errors reported about them.

Raw type's type arguments are returned A raw type's type argument is a wildcard. Statically, PICO doesn't have any knowledge what type argument is actually passed. So, PICO assumes the type argument for raw type to be "? extends @Mutable Object". If this is the return type of a method defined in raw types, then casting it to `immutable` types such as String or assigning it to `immutable` types causes warnings to be reported. For example:

```
// jdepend/src/jdepend/framework/PackageFilter.java:85
warning: [cast.unsafe] "@Initialized ?[ extends @Initialized
   @Mutable Object super @Initialized @Bottom Void]" may not be
   casted to the type
"@Initialized @Immutable String"
    public void addPackages(Collection packageNames) {
        for (Iterator i = packageNames.iterator(); i.hasNext();)
            {
            addPackage((String)i.next());
        }
    }
```

This is only a warning, because the implementation might consistently pass and get types which don't throw errors at runtime.

Calling instance methods that are not known to be only dependent on abstract state from hashCode() and equals(Object) methods

```
// jdepend/src/jdepend/framework/JavaPackage.java:266
[object.identity.method.invocation.invalid] Cannot invoke non-
   object identity method getName() from object identity context!
public class JavaPackage {
    private String name;

    public int hashCode() {
        return getName().hashCode();
    }

    public String getName() {
        return name;
    }
}
```

The hashCode() method invokes the getName() method internally, but the getName() method doesn't have a method annotation @AbstractSateOnly that specifies it's only dependent on the abstract state of the receiver, even though its implementation is only dependent on abstract state (`RDA immutable` field). Similar to the `readonly` length() method, PICO cannot infer without explicit usage of @Readonly, without explicit usage of @AbstractSateOnly, getName() is considered to be not only dependent on the abstract state, therefore calling it from hashCode() is warned.

Accessing non-abstract state fields in hashCode() and equals(Object) methods

Example:

```
// react/src/main/java/com/flowpowered/react/math/Matrix3x3.java
   :346
warning: [object.identity.field.access.invalid] Object identity
   context cannot reference non abstract state field mRows!

package com.flowpowered.react.math;
class Matrix3x3 {
    private final Vector3[] mRows = {
            new Vector3(),
            new Vector3(),
            new Vector3()
    };

     @Override
    public int hashCode() {
        int hash = 7;
        hash = 83 * hash + Arrays.deepHashCode(mRows);
        return hash;
    }
}

package com.flowpowered.react.math;
/**
* Represents a 3D vector in space.
*/
public class Vector3 {...}
```

Unannotated `Vector3` is a `mutable` type element (by default), so the type of unannotated `mRows` is `mutable Vector3 RDM []`. According to the discussion in section 4.5, `mRows` doesn't belong to the abstract state of `Matrix3x3` because the array component is `mutable` (not recursively `RDM` or `immutable`). So, accessing `mRows` in the hashCode() method, which is @AbstractStateOnly, causes errors.

5.2.2 PICOInfer

We have already seen PICO typechecker gives errors on unannotated code, because of the lack of necessary mutability annotations about which PICO can do nothing. If users manually annotate the source code, the overhead will be huge. Especially for very large projects, it would be impossible for users to inspect every piece of the program and add annotations. PICOInfer is developed to address this issue of manual annotation overhead.

However, PICOInfer doesn't solve all problems listed in section 5.2.1. Initialization related errors don't go away. Users need to manually annotate correctly with the @UnderInitialization annotation on helper methods called only from constructors; PICOInfer doesn't infer assignability qualifiers, so users also need to manually annotate some fields with @Assignable to exclude them

from the abstract state if those fields are re-assigned[4]. Inference solutions don't necessarily satisfy the requirement of @AbstractStateOnly methods because we can't currently generate a correct set of constraints in inference mode for @AbstractStateOnly methods. The reason is that for method invocations from the hashCode() and equals(Object) methods, those methods should also be annotated with @AbstractStateOnly. However, Checker Framework Inference doesn't support inferring the method declaration annotation @AbstractStateOnly nor did we manually do the job to annotate methods with @AbstractStateOnly annotations. Therefore, we didn't generate constraints that enforce that methods are really only dependent on abstract states. It would be an interesting future work to see the effect of enforcing @AbstractStateOnly on inference results in PICOInfer.

We found PICOInfer can effectively give mutability solutions for small to large projects. However, insertion of inference solutions back to the source code still have some problems, which are due to bugs in the Checker Framework Inference and annotation-tools as we discussed before.

We experimented PICOInfer on the same set of real-world projects as in the figure 5.1, and analyzed the inferred results from the following dimensions:

- Correctness: do the inferred results obey PICO's type rules?
- Usefulness: how much do the inferred results help users to understand their code? How close are the results to real-world usage?
- Performance: how fast is PICOInfer?

As we discussed before, PICOInfer is aware of the existing @UnderInitialization qualifiers in the source code, and generates more relaxed constraints. We tried manually annotating all the libraries with @UnderInitialization qualifiers in the proper locations, and compared the results to the case without the @UnderInitialization annotations. We really see the difference in inference results between completely unannotated programs and partially annotated programs with correct @UnderInitialization. The @Assignable qualifier is also manually annotated properly to allow PICOInfer to give solutions for two of the benchmark projects, jdepend and jReactPhysics3D.

Instead of only being able to infer just valid solutions for a program, we implemented PICOInfer to be tunable using PreferenceConstraints, making the solutions more useful and meaningful. PreferenceConstraint is an existing Constraint in the Checker Framework Inference, to prefer a particular solution to be inferred at a location if there can be multiple valid solutions. Each PreferenceConstraint can be attached a weight. The MaxSAT solver we use in PICOInfer, will give a set of solutions so that the sum of weights of all PreferenceConstraints are maximized. There are three locations that has PreferenceConstraints right now. They include:

- Instance Field: prefer recursively `RDM` or `immutable` at weight 3.
- Type Element Bound: prefer `RDM` or `immutable` at weight 2
- Method Declared Receiver: prefer `readonly` at weight 1
- Method Formal Parameters: prefer `readonly` at weight 1

[4]Right now, any re-assignments to initialized object is considered mutation to the receiver object because PICO and PICOInfer don't support lazy initialization at the moment. It's a future work to support lazy initialization.

Preference 1 makes more fields be in the abstract state, which is a good thing. We think this is the most important preference compared to the others, so we attach the biggest weight, 3.

Preference 2 makes more classes be `RDM` and `immutable`, because we think these classes are more valuable than trivial `mutable` classes.

Preference 3 and preference 4 prefer method declared receivers and formal parameters be `readonly`, making methods more general to be invocable and to accept more kinds of arguments. This preference avoids inferring method declared receivers and formal parameters only based on current invocations, and propagating the same types as current invocations. This is the weakest preference we need compared to the below two preference, so we attach weight 1.

Note that there isn't the best weighting schema. Weights should be adjusted depending on what are the most important properties that a user want. We compared the difference between using the preference mechanism with the no-preference case, and got the result that adding preferences did improve the accuracy and usefulness of inference solutions.

Only allowing upcasts in inference is too restrictive. Downcasts should also be allowed to infer annotations for more projects. PICOInfer supports three cast policies, in which Comparable Cast is the default. For example, in

```
(@1 A) @2 B
```

- Only Upcast: generate subtype constraint @2 <: @1
- Comparable Cast: generate ComparableConstraint @1 <:> @2.
- Any Cast: doesn't generate any constraint. @2 can be directly casted to @1. One can always upcast to a super type and then downcast to a subtype. But if programmers didn't split these operations to two separate casts (one upcast and one downcast), it would be not possible to achieve the same effect as Any Cast using only one cast expression.

If the to-be-inferred program contains casts that are annotated with mutability qualifiers by users and the user-written casts are not safe under the selected cast policy at compile time, PICOInfer doesn't exit the inference process. Instead, PICOInfer will automatically apply Any Cast in those cast locations to respect the existing programmer-annotated casts. After the inference, if users use PICO type checker to typecheck the program, PICO type checker still reports a warning for it. The motivation for this is to make more programs inferrable. Again, since PICO and PICOInfer is a static type system that doesn't have a runtime component, we need to find a balancing point between type soundness and freedom. It would be user's decision which cast policy to choose according to their specific requirements and goals.

It's inevitable for a project to interact with third party code whose source code is not checked nor inferred by PICO. A stub file is a file that specifies annotations on type elements and method signatures so that they can be picked up in typechecking or inference instead of applying defaults for them. Although users can supply stub files that specify the signatures of methods from those third-party code, it would not be practical to list all of them. PICOInfer has to default qualifiers on some of the unchecked third-party methods. There can be three kinds of defaulting philosophies: pessimistic, realistic, and optimistic:

- Pessimistic assumption: every unchecked method returns `readonly` objects and accepts `bottom` arguments.
- Realistic assumption: every unchecked method accept `mutable` arguments and returns `mutabl e` objects.
- Optimistic assumption: every unchecked method accepts `readonly` arguments and returns `bottom` objects.

The pessimistic assumption would cause every inference to fail, because neither `mutable` or `immutable` objects can be passed into third-party methods anymore. Returned objects are `readonly`, therefore any mutations to them would cause inference to give no solutions.

The realistic assumption defaults `mutable` to every unchecked third-party method parameter and return type. For parameters that are actually only read, defaulting them to `mutable` would cause inference to give no solutions if `immutable` arguments are passed in. For example, when `immutable` Strings are passed to a logger method with java.lang.Object parameters only for printing them, PICOInfer can't give solutions because PICOInfer assumes the parameters are mutated in the unchecked method. But due to the fact that objects in OO programs need to change states frequently, this realistic `mutable` default makes sense.

Other than the above two, in the optimistic assumption, methods have `readonly` parameters, so they accept all kinds of arguments (either `mutable` or `immutable`), and return `bottom` objects so the methods are invocable on every site. This is the upper bound of all possible inference solutions in terms of `readonly` solutions.

At the implementation level, there is a flag `-AuseDefaultsForUncheckedCode=bytecode` that does the job. @Readonly and @Bottom annotations can be declared with either the pessimistic or optimistic assumption. Then passing the above flag to PICOInfer will tell PICOInfer to use the configured defaults in declarations of @Readonly and @Bottom for third-party library methods from bytecode (whose source code is unavailable). If no such flag is passed, the realistic defaulting assumption is used.

Real-world programs fall into the range between the realistic assumption and the optimistic assumption. Solutions for those programs must be bounded between the two boundaries. The pessimistic assumption is so conservative that no programs can invoke any unchecked methods from third-party libraries including the Java JDK. It would be interesting to see what inference results are in each of the cases and have an understanding about the ranges of possible solutions.

5.2.2.1 Inference Result Manual Inspection

As the first step, we conducted experiments on the 14 projects. We manually inspected part of the source code after inference to see whether PICOInfer gives correct and useful solutions for them.

In this section, in order to get solutions, and to get more useful solutions, benchmark projects that have initialization errors are manually annotated with @UnderInitialization annotations. Fields that are re-assigned in `readonly` instance methods (invoked from the hashCode() method, for example) are annotated with @Assignable. Fields that belong to an `immutable` object used as map keys but are outside the abstract state also get manual @Assignable annotations. We use default comparable cast to generate constraints on casting sites. We chose the realistic defaulting

assumption and enabled PreferenceConstraint for better solution quality. We believe the above settings can offer the best solutions for us to analyze their correctness and usefulness.

Figure 5.3 shows details about how many manual annotations are added to each project.

Project Name	Number of @UnderInitialization	Number of @Assignable
ECC-RSA-Backdoor	2	0
jump	1	0
jdepend	5	1
jama	5	0
jdeb	8	0
jReactPhysics3D	1	19
react	1	0
jblas	8	0

Figure 5.3: Number of @UnderInitialization and @Assignable added

Adding @UnderInitialization qualifiers manually doesn't indicate that all the FBC type errors are gone, but at least we know that FBC errors that affect the inference results of PICOInfer will be gone. This guarantee is already good enough for better results for mutability.

Only jdpend and jReactPhysics3D must have @Assignable annotations added so that PICOInfer can give solutions.

jdepend also uses raw typed HashTable instead of the latest generics in Java. Statically, PICOInfer can only assume the type argument for raw types are "? extends @Mutable Object". Some methods do return the wildcard type argument, and cast them to `immutable` String or assign the returned vaule to `immutable` String. However, the implementation of jdepend correctly casts wildcard type arguments consistently with element types when being put into the HashTable. In order to infer this project, PICOInfer always enables Any Cast. Otherwise, there won't be solution.

Figure 5.4 lists the number of each qualifier in inference solutions.

Project Name	Readonly	Immutable	RDM	Mutable	Bottom
bdoorProto	33	101	2	29	1
AFHbackdoor	16	49	6	38	11
FaceDetection	227	342	22	247	77
imgscalr	83	217	3	355	49
ECC-RSA-Backdoor	736	387	17	1226	178
jump	990	699	1	243	51
jdepend	736	610	102	1966	352
jama	287	1613	74	390	409
exp4j	330	823	34	277	112
jdeb	796	1390	130	1905	311
jReactPhysics3D	3950	2360	503	7564	640
react	4007	2962	513	5829	567
JLargeArrays	2039	4388	145	3197	724
jblas	7404	7466	113	5133	645

Figure 5.4: Number of each qualifier in the best combination of settings

Below is a class in project jdeb after inference:

```
/**
* Builds the Debian changes file.
*/
@Immutable
class ChangesFileBuilder {

    public @Mutable ChangesFile createChanges(
      @Readonly ChangesFileBuilder this,
      @Readonly BinaryPackageControlFile packageControlFile,
      @Readonly File binaryPackage,
      @Readonly ChangesProvider changesProvider) throws
         IOException, PackagingException {

        ChangesFile changesFile = new @Mutable ChangesFile();
        changesFile.setChanges(changesProvider.getChangesSets())
           ;
        changesFile.initialize(packageControlFile);

        changesFile.set("Date", ChangesFile.formatDate(new
           @Mutable Date()));

        try {
          // compute the checksums of the binary package
          InformationOutputStream md5output = new @Mutable
             InformationOutputStream(new @Mutable
             NullOutputStream(), MessageDigest.getInstance("MD5")
             );
          InformationOutputStream sha1output = new @Mutable
             InformationOutputStream(md5output, MessageDigest.
             getInstance("SHA1"));
          InformationOutputStream sha256output = new @Mutable
             InformationOutputStream(sha1output, MessageDigest.
             getInstance("SHA-256"));

          FileUtils.copyFile(binaryPackage, sha256output);

          changesFile.set("Checksums-Sha1", sha1output.
             getHexDigest() + " " + binaryPackage.length() + " "
             + binaryPackage.getName());

          changesFile.set("Checksums-Sha256", sha256output.
             getHexDigest() + " " + binaryPackage.length() + " "
             + binaryPackage.getName());

          StringBuilder files = new @Mutable StringBuilder(
```

```
            md5output.getHexDigest());
          files.append(' ').append(binaryPackage.length());
33        files.append(' ').append(packageControlFile.get("
            Section"));
          files.append(' ').append(packageControlFile.get("
            Priority"));
35        files.append(' ').append(binaryPackage.getName());
          changesFile.set("Files", files.toString());
37      } catch (@Mutable NoSuchAlgorithmException e) {
          throw new @Mutable PackagingException("Unable to
            compute the checksums for " + binaryPackage, e);
39      }

41      if (!changesFile.isValid()) {
          throw new @Mutable PackagingException("Changes file
            fields are invalid " + changesFile.invalidFields() +
43    ". The following fields are mandatory: " + changesFile.
        getMandatoryFields() +
      ". Please check your pom.xml/build.xml and your control file
        .");
45      }

47      return changesFile;
      }
49  }
```

- All method parameters and receivers are inferred to be @Readonly. There are no mutating methods called on them or re-assignments to their states. All the existing accesses to them are readonly operations.
- The initializer for the local variable `changesFile` is inferred to be @Mutable, because in the method body, there are many setters called on it, such as setChanges(), initialize() etc. The method return type is also @Mutable, which is consistent with the fact that `changesFile` is returned in the line 47.
- Since class ChangesFileBuilder is just a builder that doesn't have state, its bound is inferred to be @Immutable, instead of @Mutable.
- The createChanges() method gets a @Readonly receiver, because its implementation only allocates a new object, mutates it and returns it without affecting the abstract state of the ChangesFileBuilder instance. A @Readonly declared receiver is better than @Immutable declared receiver because it is more general.

We could find the inferred solutions are:

- correct: There are no mutations to objects referred by `readonly` references

- useful: Solutions didn't just contain conservative types, but instead properly assigned mutability types that reflect real-world needs. For example, even though the builder class is inferred to be @Immutable, the declared receiver of the createChanges() method still gets inferred with @Readonly, which is more general than @Immutable.

Manually inspecting all the results is inpractical. We tried to run PICO type checker on the inferred projects to check inferred results. However, then are several factors that make them fail to typecheck:

- Annotation-tools throws exceptions on 9 of the projects due to its bug. Workaround for the bug inserts solutions to wrong locations in one project (we didn't investigate every project), so we gave up the workaround.

- Some annotations are inferred as solutions in the internal state of the Checker Framework Inference, but didn't get written to jaif files because of some bugs in the Checker Framework Inference. Jaif files are the instructions for our too, annotation-tools, to insert solutions back to the source code. So, some locations miss annotations that are different from the default for the corresponding locations.

- Annotation-tools doesn't support inserting annotations to some source code locations even if jaif files contain valid annotations for those locations. Even though solutions contain correct annotations, they aren't actually inserted to source code. Annotation-tools complains that those locations are not supported and skip inserting them.

So we inspected some parts of the inferred results manually according to each type rule discussed in section 4.9 and found they obey PICO type rules. We believe PICOInfer can correctly infer typechecking solutions internally. In order to verify this belief, we did try manually inferring and inserting missing annotations. By inferring, it means we examine several invocation sites or contexts of the corresponding fields and methods after inference and guess what PICOInfer may have inferred to those positions. Then we verify the inferred annotations by checking all the invocation sites and compare them against PICO type rules. Since this process can be very time-consuming, we only tried it on one project, jama. For jama, we ran PICO type checker to check whether correct annotations are "patched". The result shows that it completely typechecks under PICO type checker after being inferred+inserted by Checker Framework Inference and manual insertion of missing annotations. Its annotated result is publicly available on the GitHub repository: `https://github.com/topnessman/jama/tree/fully-annotated`.

For the following sections, the best combinations of settings are always the base case. Only one configuration changes at each section to really show the effects of that configuration change.

5.2.2.2 Inference Solutions Without Manual @UnderInitialization

Figure 5.5 shows the number of each qualifier if @UnderInitialization is not properly inserted manually.

Project Name	Readonly	Immutable	RDM	Mutable	Bottom
bdoorProto	33	101	2	29	1
AFHbackdoor	16	49	6	38	11
FaceDetection	240	302	32	264	77
imgscalr	83	217	3	355	49
ECC-RSA-Backdoor	738	383	22	1220	181
jump	990	699	1	243	51
jdepend	724	615	84	1990	353
jama	306	1601	69	386	411
exp4j	331	819	36	278	112
jdeb	796	1390	130	1905	311
jReactPhysics3D	3946	2359	503	7564	645
react	4176	3052	504	5582	564
JLargeArrays	2039	4388	145	3197	724
jblas	7418	7605	131	4830	634

Figure 5.5: Number of each qualifier without manual @UnderInitialization annotations inserted

We expected the number of @Mutable annotations will increase if source code is not annotated with @UnderInitialization, because there will be more restrictive constraints to make references be @Mutable, for example an instance method that assigns a field. Those will be inferred to @Mutable. However, the actual data indicates this is not the case.

We did see a change in the inferred result in ECC-RSA-Backdoor: Before removing @UnderInitialization annotation on the copy() method, its declared receiver is inferred to @Immutable. However, after removing @UnderInitialization, its declared receiver is inferred to @Mutable. Figure 5.6 and Figure 5.7 show each of the cases respectively.

```
1 private void copy(@UnderInitialization @Immutable BinNum this,
    @Mutable ALB num){
          this.num = new @Mutable ALB().addA(num);
3      }
```

Figure 5.6: Inferred results with @UnderInitialization

```
1 private void copy(@Mutable BinNum this, @Mutable ALB num){
          this.num = new @Mutable ALB().addA(num);
3      }
```

Figure 5.7: Inferred results without @UnderInitialization

5.2.2.3 Inference Solutions Without Manual @Assignable

Only two projects, jdepend and jReactPhysics3D, are affected by removing manual @Assignable annotations. They failed to be inferred.

For jdepend, the failure reason is that a `readonly` instance method getChildren() re-assigns the children field. Removing @Assignable makes the children field @RDA, and thus it cannot be re-assigned through `readonly` references. PICOInfer detected such violation of PICO type rules, and failed to give solutions for jdepend.

```
class AfferentNode extends PackageNode {
  ...
  public String toString() {
    if (getParent() == null) {
      return "Used By - Afferent Dependencies" + " ("+
         getChildren().size() + " Packages)";
    }

    return super.toString();
  }
  ...
}

class PackageNode {
  ...
  public ArrayList getChildren() {
    if (children == null) {
      children = new ArrayList();
      ... Some initialization code goes here ...
    }

    return children;
  }
  ...
}
```

One might argue that the `children` field is part of the abstract state of the PackageNode object. The getChildren() method is nothing else but lazy initialization of the field, but not arbitrary re-assignment. PICO right now doesn't support lazy initialization: it assumes that for initialized objects, assignments to their fields are mutations. It is a future work to let PICO support lazy initialization for better typechecking errors and inference results.

For jReactPhysics3D, RigidBody objects are used as map keys, but are re-assigned fields after being initialized. Removing @Assignable annotations from those fields make them be in the abstract state of the RigidBody objects, therefore PICOInfer cannot infer solutions for them. This would be really a problem if the hashCode() method was overriden to depend on those re-assigned fields. So, PICOInfer catches this possible bug by failing to give solutions for this project.

If all third-party library methods are properly annotated with mutability qualifiers in the stub file, and PICOInfer still cannot infer solutions, it indicates that the project won't have the guarantees that PICO provides. In this case, programmers should go back to the source code and add/change annotations in order for the program to be inferred by PICOInfer, and gain confidence about the their code using PICO.

5.2.2.4 Inference Solutions With Optimistic and Pessimistic Assumptions

Project Name	Readonly	Immutable	RDM	Mutable	Bottom
bdoorProto	44	103	2	16	1
AFHbackdoor	28	49	6	30	17
FaceDetection	296	418	12	91	88
ECC-RSA-Backdoor	894	458	25	985	182
jump	977	760	4	185	58
jama	291	1573	19	481	410
exp4j	357	909	28	152	113
jReactPhysics3D	3785	2874	452	7228	632
react	4312	3085	500	5364	550
jblas	6127	7189	171	6506	644

Figure 5.8: Number of each qualifier with optimistic assumption for unchecked methods

Figure 5.8 shows the number of each qualifier when using the optimistic assumption for unchecked third-party code. This is the other bound for each qualifier compared to the realistic assumption. The best-case solutions fall in this region: min of the corresponding qualifier in figure 5.4 and max of corresponding qualifier in figure 5.8. In pessimistic assumption for unchecked methods, all projects fail to get solutions. These bounds are trivial: all 0s, therefore we don't consider them. Using the range of each qualifier, we can get a rough idea about where the most-suitable solutions are.

5.2.2.5 Inference Solutions Without PreferenceConstraint

Figure 5.9 shows number of each qualifer without PreferenceConstraints.

Project Name	Readonly	Immutable	RDM	Mutable	Bottom
bdoorProto	10	89	22	41	2
AFHbackdoor	2	58	5	40	15
FaceDetection	81	435	52	267	79
imgscalr	19	262	0	373	52
ECC-RSA-Backdoor	252	1455	145	475	217
jump	774	902	5	250	53
jdepend	455	718	138	2095	360
jama	42	1765	105	439	422
exp4j	109	991	53	312	111
jdeb	349	1528	154	2193	307
jReactPhysics3D	1306	3195	493	9402	620
react	1654	3261	502	7867	593
JLargeArrays	879	5253	47	3545	768
jblas	1621	11526	697	5981	788

Figure 5.9: Number of each qualifier without PreferenceConstraints

Compared to the results in figure 5.4, if PreferenceConstraints are disabled, the number of @Mutable and @Bottom qualifiers increased in every project. These increases indicate trivial @Mutable and @Bottom are inferred more, which means the inferred results become less useful. @Readonly annotation' numbers decreased, meaning less method parameters and declared receivers are inferred to be @Readonly, which is a bad thing, because inferring them to @Readonly allows more arguments to be passed in. Numbers of @Immutable and @ReceiverDependentMutable didn't necessarily go up or down. They don't have a fixed pattern to predict their increase or decrease. In order to get more useful inference results, it's a must to enable PreferenceConstraints.

5.2.2.6 Checker Framework Inference Statistics With And Without PreferenceConstraint

Checker Framework Inference statistics include how many slots and constraints are generated. In PICOInfer, there are always 5 ConstantSlots generated, which corresponds one-to-one to the mutability qualifiers — `readonly`, `mutable`, `RDM`, `immutable` and `bottom`. Numbers of constraints differ between enabling preferences and disabling preferences.

Figure 5.10 shows Checker Framework Inference statistics without and with preference constraints.

Project Name	Slot	Total Constraint		PreferenceConstraint	
		NoPreference	Preference	NoPreference	Preference
bdoorProto	177	776	837	0	61
AFHbackdoor	126	769	819	0	50
FaceDetection	940	4357	4551	0	194
imgscalr	715	5150	5332	0	182
ECC-RSA-Backdoor	2560	11603	11759	0	156
jump	2012	12321	12733	0	412
jdepend	3689	17968	18632	0	664
jama	2839	8464	8727	0	263
exp4j	1630	7733	8088	0	355
jdeb	4613	28067	29232	0	1165
jReactPhysics3D	15607	62197	64591	0	2394
react	14308	63757	66422	0	2665
JLargeArrays	10707	41897	43574	0	1677
jblas	20688	97462	103188	0	5726

Figure 5.10: Numbers of Slots and Constraints (and PreferenceConstraints) generated

The same numbers of Slots are generated for each project without and with PreferenceConstraints. Enabling PreferenceConstraint only adds the number of PreferenceConstraint but doesn't affect the numbers of other constraints.

5.2.2.7 Solver and Timing Statistics With and Without PreferenceConstraint

In order to find out how fast the Sat4J solver can solve the constraints with the size of projects growing, we mainly tracked three different kinds of time: parsing time for Checker Framework Inference to parse the source file, create Slots and generate Constraints over the Slots; encoding time for encoding Constraints into solver domain-specific problems; and solve time for solvers to really solve the domain-specific problems. For timing statistics, we run each experiment 3 times and take their average to reduce deviations. Figure 5.11 illustrates this.

From figure 5.11, we can find that numbers of CNF variables are the same with and without PreferenceConstraint. This is expected, because PreferenceConstraint doesn't add new variables — it only adds additional CNF clauses to let solver prefer a particular solution. This also explains why the number of CNF clauses increases when PreferenceConstaint is enabled.

The timing results are a little more interesting. Some of the results are counter-intuitive, because normally having extra PreferenceConstraints to solve seems to slow down solving speed. However, the experiment results show that this seemingly obvious conclusion isn't true. In projects such as jReactPhysics3D and react, solving time decreases even with PreferenceConstraints. This might be due to some optimizations in Sat4J library. Overall, it at least indicates that having PreferenceConstraints won't slow down solving speed too much.

We can also notice that parsing time takes the most portion of the inference time. As the size of projects go up, parsing times grow much faster than encoding and solve time. Parsing time is the bottleneck of the performance of PICOInfer. However, parsing is done by Checker Framework Inference, so PICOInfer cannot really do anything to improve its performance without speeding up parsing time for inference.

Overall, encoding and solving times are very small even the project sizes grow a lot (less than 5 seconds). The sum of encoding time and solving time doesn't grow as fast as the CNF variable sizes and CNF clauses sizes. From bdoorProto to jblas, the number of CNF variables grows more than 100+ times, but the sum of encoding and solving time only grows less than 3 times. This means Sat4J library is extremely scalable as the projects sizes grow.

The hardware and software environment for the experiments is:

- Personal laptop
- Intel Core i5-7200U CPU @ 2.50GHz 4 processor
- 7.7 GB RAM memory
- 64-bit Ubuntu 16.04 OS
- Java(TM) SE Runtime Environment (build 1.8.0_73-b02)
- Max Heap Size: 2.07 GB

This environment information also applies to experiments of GUTInfer in section 6.4.

Project Name	CNF Variable	No Preference				Preference			
		CNF Clauses	Timing(ms)			CNF Clauses	Timing(ms)		
			Parsing	Encoding	Solve		Parsing	Encoding	Solve
bdoorProto	2317	3801	2211	14	1014	3884	2273	20	1015
AFHbackdoor	1643	2820	2405	12	1009	2870	2447	12	1014
FaceDetection	12509	21146	5307	47	1004	21340	5326	48	1010
imgscalr	10073	18341	7252	41	1005	18523	5871	38	1003
ECC-RSA-Backdoor	35593	64515	10785	89	1009	64671	9942	91	1005
jump	28757	54001	6445	104	1005	54413	5995	126	1007
jdepend	49660	85619	10635	101	1003	86283	10930	107	1005
jama	37794	60959	18543	84	1012	61222	17194	67	1008
exp4j	21563	37378	6697	62	1009	37733	6093	55	1002
jdeb	64601	115472	13565	135	1002	116635	13440	196	1006
jReactPhysics3D	216620	384064	27852	442	4005	386456	26025	317	2004
react	196895	359351	32409	319	4008	362014	35423	351	2003
JLargeArrays	148779	251445	34160	196	1002	253120	32686	236	1411
jblas	292800	515254	60613	423	2003	521035	56907	565	1002

Figure 5.11: Solver related statistics and timing information with and without PreferenceConstraint

Chapter 6

Improvements to the Generic Universe Type System

This chapter is structured as follows: section 6.1 gives implementation-level information about the improvements to GUT type system. Section 6.2 discusses additional types that are made implicitly `bottom`. Section 6.3 discusses possible problem of viewpoint adapting types to `bottom` receiver and solution for it. Section 6.4 shows experiment results of running GUTInfer on real-world projects.

6.1 Implementation Improvements

GUT and GUTInfer were implemented based on the Checker Framework and Checker Framework Inference, but typechecking checker and inference checker were seperate before. This caused extra effort to maintain the two systems and to keep them in sync. We unify the typechecking and inference system for GUT in one checker. The new checker can both typecheck programs and infer annotations for unannotated programs. Existing test cases all passed after this refactoring.

The new checker uses framework-level viewpoint adaptation logic that greatly reduces the amount of code from GUT implementation. On the typechecking side, GUT only needs to specify a minimum set of information to specify how GUT qualifiers can be combined and what result each combination yields. On the inference side, GUTInfer only overrides InferenceViewpointAdapter for the sake of performance. For example, if the declared type is implicitly `bottom`, the result should always be `bottom` so there is no need to use the general implementation in InferenceViewpointAdapter to generate a new result Slot — keeping the declared type is enough.

Previously, GUTInfer had Sat4J[12] encoding logic of viewpoint adaptation in its own package. We decided to change to the new Type Constraint Solver[21], because it can save lots of duplicate work in GUT. For example, SubtypeConstraint, EqualityConstraints and so on are common and don't need to be implemented inside GUT again. The only thing we can't get for free from the Type Constraint Solver is the encoding logic for viewpoint adaptation, as it needs type-system-specific logic. We adapted the logic of encoding CombineConstraints based on the new Type Constraint Solver.

The new implementation contains 17 Java files and 1478 lines of Java code for core logic; 37 files and 654 lines of Java code for the testcases.

In order to differentiate GUT type checker and GUT inference, we still use GUTInfer to indicate the checker is used in inference mode.

6.2 Implicit Bottom Types

In GUT, there is an internal qualifier `bottom` that means no ownership is needed. Primitive types such as int, long, and float can only hold numeric values that aren't owned by any object. So, primitive types have type `bottom`.

However, the previous GUT implementation doesn't handle boxed primitive types and String types well. They can have arbituary ownership modifiers on them and those ownership modifiers don't have any effect on the subtype relations. For example, the below are true in old GUT:

```
@Rep String <: @Peer String
@Any Integer <: @Rep Integer
@Rep Integer <: @Bottom int
```

Ownership modifers are inconsistently used for boxed primitive types and Strings, and cause confusion. Especially on the inference side, constraint generations on them are confusing.

To address the inconsistency problem, we changed implicit rules so that all the below types and literals are implicitly `bottom`:

- Primitive types: int, double, boolean, byte, char, float, int, long and short
- Boxed primitive types: Integer, Double, Boolean, Byte, Character, Float, Integer, Long, Short
- String
- All literals: primitive literals, null literal, class literals

If other ownership qualifers are used on the above types, a type-invalid error will be raised.

6.3 Viewpoint Adaptation To Bottom Receiver Problems

After making the eight boxed primitive types and java.lang.String implicitly `bottom`, there comes another problem: parameters of instance methods declared inside those classes can't be defaulted to `peer` anymore. For example:

```
package java.lang;
public final class String {
  boolean contentEquals(/*@Peer*/ StringBuffer sb) {...}
}
...
String s = "Hello";
s.contentEquals(...);
```

Method invocation at line 7 will always get stuck: by default, parameter `sb` has type @Peer StringBuffer. Since method invocation on line 7 is an instance method invocation, the declared type of the parameter, @Peer, should be adapted to @Bottom String, which results in @Lost according to the viewpoint adaptation rules of GUT. However, writing to a `lost` type is not allowed at any time in GUT as we discussed before. Therefore, no matter what arguments are passed, the method invocation itself will always get stuck. In order to fix this issue, we manually annotated methods declared in the boxed primitive types and String: all reference typed parameters (except implicitly `bottom` typed ones) are annotated to accept `any` arguments in the stub file. Constructor invocations also have the same problem. We also manually annotated their parameter types to `any`.

The reason why we make the parameters `any` isn't just because after viewpoint adaptation they yield non-`lost` results, but we also believe they are not mutated inside the method bodies and constructor bodies. Besides, the parameters of the methods and constructors don't need concrete ownership type information. Based on the above two reasons, we make the parameters `any`.

6.4 Experiments - GUTInfer

6.4.1 Benchmarks

We chose 10 open-source projects — four benchmarks from previous GUT inference paper[19], SRI[1] hackathon projects, and some scientific libraries used by researchers and scientists from GitHub to evaluate how GUTInfer performs on real-world projects. Some of them are the same as ones in PICO's experiments in section 5.2. Below is a summarization of additional libraries that are not introduced before in section 5.2.

Project Name	File	Blank	Comment	Sloc
bdoorProto	8	52	81	152
AFHbackdoor	5	49	26	161
imgscalr	11	302	2186	1181
ECC-RSA-Backdoor	5	416	332	1251
javad	32	922	1410	1846
jdepend	22	869	942	2557
jama	10	589	931	2599
classycle	87	659	3540	4678
zip	48	956	5266	5466
JLargeArrays	17	882	2496	11337

- **javad** a Java class file disassembler
- **classycle** analysing tool for Java class and package dependencies
- **zip** JDKs implementation of the zip and gzip compression algorithms, taken from OracleJDK 1.7

[1] https://www.sri.com/

6.4.2 Inference Results

The numbers of peer and `rep` qualifiers inferred when enforcing static topology only and enforcing the owner-as-modifier principle is shown in figure 6.1; the numbers of `any`, `lost`, `self`, `bottom` qualifiers inferred when enforcing static topology only and enforcing the owner-as-modifier principle is shown in figure 6.2;

Project Name	Peer		Rep	
	Topol	OAM	Topol	OAM
bdoorProto	4	7	23	22
AFHbackdoor	22	23	1	0
imgscalr	175	172	4	7
ECC-RSA-Backdoor	1034	1107	24	24
javad	212	424	170	115
jdepend	792	1027	371	294
jama	146	282	223	312
classycle	1040	1260	608	595
zip	436	514	248	209
JLargeArrays	790	902	1354	1458

Figure 6.1: Numbers of `peer` and `rep` qualifiers inferred for static topology and owner-as-modifier

Project Name	Any		Lost		Self		Bottom	
	Topol	OAM	Topol	OAM	Topol	OAM	Topol	OAM
bdoorProto	38	36	3	3	29	29	43	43
AFHbackdoor	11	12	21	20	24	25	42	41
imgscalr	174	175	88	87	50	50	227	227
ECC-RSA-Backdoor	262	260	690	644	202	186	309	300
javad	270	163	190	154	250	235	612	613
jdepend	487	314	655	616	743	712	677	631
jama	874	669	204	198	156	153	955	944
classycle	816	652	935	892	780	785	1491	1486
zip	294	250	359	361	473	474	900	902
JLargeArrays	1475	1445	1307	1145	1148	1124	3294	3294

Figure 6.2: Numbers of other ownership qualifiers inferred for static topology and owner-as-modifier

GUTInfer can infer solutions that enforce only the static topology or the owner-as-modifier principle. In the static topology mode, GUTInfer only infers topology of objects, but doesn't prevent non-pure methods from being invoked on non-owner references. Pure methods are methods that don't side effect receiver objects' states. In the owner-as-modifier mode, however, solutions guarantee that `any` and `lost` references are not used to call non-pure methods because `any` and `lost` references mean the current receiver object is not the owner of the referred-to object. We can see from figure 6.2 that after enabling the owner-as-modifier principle, numbers of `any` and `lost` references go down, which is expected. Figure 6.1 shows that numbers of `peer` go up, but numbers of `rep` references go down in general. This is normal, because if there are more numbers

of ownership levels, more easily `lost` is produced. But `lost` is forbidden to be used mutate the receiver object. So ownership structure becomes flatter (more `peer`). But we also see in some projects, numbers of `peer` and `rep` both increase. This may be because there were too many `any` and `lost` references that were used to mutate the receivers before. Having more `rep` in this case means that previous no-so-structured `any` and `lost` aliases now form a hierarchical structure, so the mutations are initiated more by owners.

For the correctness of inference results, we choose 1 project imgscalr and use GUT type checker to check whether the inferred results typecheck or not. The result shows that both inference solutions for enforcing the static topology only and enforcing the owner-as-modifier principle typecheck.

6.4.3 Checker Framework Inference Statistics

We collected the number of slots and constraints generated for each project. In each project, there are always 6 ConstantSlots generated, each for one GUT qualifier. Generally, as the sizes of projects grow, numbers of slots and constraints increase. Figure 6.3 shows how many slots are generated for each project and numbers of constrains when enforcing the static topology and enforcing the owner-as-modifier principle.

Project Name	Slot	Constraint	
		Topol	OAM
bdoorProto	163	300	317
AFHbackdoor	131	271	309
imgscalr	753	1913	1973
ECC-RSA-Backdoor	2570	5285	5756
javad	1815	3986	4186
jdepend	3805	6364	6705
jama	2836	5521	5756
classycle	5901	10852	11654
zip	2764	6221	6585
JLargeArrays	10060	22791	24035

Figure 6.3: Numbers of Slots and Constraints generated

When enforcing the owner-as-modifier principle, in addition to the constraints generated for static topologies, additional InequalityConstraints are generated to forbid modification through `any` or `lost` references. Therefore, increases in the number of constraints are as expected.

6.4.4 Solver and Timing Statistics

Figure 6.4 lists solver-related statistics and timing statistics for each project when only enforcing the static topology. For the running time, we run each project three times and take the average to reduce the deviation.

Project Name	CNF Variable	CNF Clauses	Timing(ms)		
			Parsing	Encoding	Solve
bdoorProto	2326	4338	2280	10	1010
AFHbackdoor	1779	3336	3563	15	1016
imgscalr	11469	21360	6338	39	1007
ECC-RSA-Backdoor	38747	87488	11380	84	2349
javad	28067	56456	5935	62	1004
jdepend	51491	104666	9188	104	1342
jama	44458	83964	17862	75	2007
classycle	86408	173719	13256	118	2670
zip	42037	83866	7867	76	1385
JLargeArrays	152953	299988	38352	221	9008

Figure 6.4: Solver statistic and timing information for static topology

Figure 6.4 shows that as the sizes of projects grow, sizes of CNF variables and clauses grow quite fast. Most of the time in inference is spent on parsing Java source files, generating slots and constraints. Encoding time and solve time grow much slower than the growing speed of CNF variables and clauses.

Hardware and software information is the same as that in experiments for PICO. See section 5.2.2.7 for details.

Chapter 7

Problems And Future Work

Although we successfully applied GUT and PICO on some real-world projects, there is still lots of space for improvement.

First, GUTInfer still infers @Self and @Bottom qualifiers on locations that are not qualified to use those two annotations. But this problem is a general issue for every type systems that uses Type Constraint Solver. The reason is that Type Constraint Solver doesn't consider the semantics of each type system and provides the most general encoding for slots, thus every qualifier has the equal opportunity of being inferred even though for that location some qualifiers are not qualified. One possible solution is to enforce a @QualifiedLocations meta-annotation, and uniformly enforce it in Checker Framework and Checker Framework Inference. On the typechecking side, if an annotation is not qualified on specific locations or types, issue errors. On the inference side, generate InequalityConstraints so that unqualified annotations are not inferred on that location or type. We have started working on this, but still need some time to implement it correctly.

Inference results right now cannot typecheck completely. One reason is as we mentioned above — some unqualified qualifiers are inferred for unqualified locations or types. The other reason is annotations are not inserted to some locations and they didn't get solutions. Even if there are solutions for some locations, they are not supported or failed to be inserted back to the source code by annotation-tools for various unknown reasons. Important future work is to file those bugs and fix them because every type system inference gets affected by them. After that, we need to run PICO type checker to verify the inferred solutions completely typecheck.

As we discussed before, inference of mutability qualifiers can be affected by existing FBC qualifier, @UnderInitialization — if a method's declared receiver type is @UnderInitialization, then field assignments in the method won't generate constraint that the receiver must be @Mutable, because during the construction phase, `immutable` and `RDM` receivers are also allowed to assign fields. However, other FBC qualifiers won't affect the inference result of mutability qualifiers. It's the best for PICOInfer to infer both FBC qualifiers and mutability qualifiers for unannotated programs. However, the Checker Framework Inference does not support inferring two type hierarchies at one time, so we can't implement PICOInfer right now to infer both initialization hierarchy and mutability hierarchy at the same time. The workaround we took in the previous section is that we manually annotated the programs with @UnderInitialization. Instead of inferring two hierarchies at the same time, another possible solution is to infer in two passes: first infer initialization hierarchy, then infer the mutability hierarchy. However, there is no efficient inference for FBC type system and it would be not trivial to implement it correctly. Again, the annotation-tools bugs

should be fixed first in order to make the two-passes approach work. Otherwise, even if correct annotations are inferred, if they are not inserted correctly, the second pass still gets an ill-typed program that biases the inference for mutability hierarchy. It's also good future work to infer assignability qualifiers.

At the moment, even though PICO can warn about depending on non-abstract-state fields in abstract-state-only methods such as hashCode(), it's not guaranteed to prevent solutions from depending on non-abstract-state fields when inferred programs are typechecked. The reason is that in practice, other instance methods may also be called from those abstract-state-only methods, but they are not annotated with meta-annotation @AbstractStateOnly. PICOInfer can only assume those methods are not only dependent on abstract states if the method called doesn't have explicit @AbstractStateOnly annotation, and abort the inference. This would be too restrictive. If we could have a way of inferring @AbstractStateOnly during the whole inference process, there could be nice solutions that prevent depending on non-abstract-state fields. However, till now, Checker Framework Inference does not support inferring method declaration annotations. We also need to think about the interactions between inference of method declarations annotations and inference of existing annotated types. It would be a good feature to support in the future. Another possible solution is to manually examine the programs and annotate methods that should be @AbstractStateOnly. But, this approach may need too much human effort.

PICO right now doesn't support lazy initialization, but it's also interesting to let it support lazy initialization, without having to make the lazily-initialized fields be @Assignable, which makes less fields be in the abstract state.

As a future work, we also want to extend the PICO formalization to also consider generics. After extending formalization to generics, we plan to prove the soundness of PICO using theorem provers such as Coq to mathematically have more confidence on PICO.

We also plan to work on a new type system that combines GUT and PICO. Having ownership and immutability type systems combined into one type system must have very interesting applications. We can use ownership information and mutability of the object itself to have many interesting combinations that help controlling side-effects through references.

Chapter 8

Conclusions

In this thesis, we presented work that makes context sensitivity a framework-level feature in the Checker Framework and Checker Framework Inference, and its successful applications to type-checking/inference and two type systems, GUT and PICO. If type systems need either one of or both receiver-context and assignment-context sensitivity, they can get it/them for free without implementing it/them again.

We also introduced a brand new immutability type system, PICO, that supports abstract, transitive, and object immutability features. We utilized the existing FBC type system to handle initialization problems nicely, without worrying about initialization of immutable objects and possible aliasing problems. We believe PICO stands out among other immutability type systems by having flexible enough features such as supports for generics, exclusion of fields from the abstract state, sensitivity to receiver contexts and assignment contexts etc. In order to reduce the burden of manually writing mutability qualifiers, we developed PICOInfer, and inferred mutability qualifiers for real-world projects up to 63.5kSloc.

We adapted the GUT implementation to the new framework-level context sensitivity feature and latest Type Constraint Solver in Checker Framework Inference, and improved its ability to better handle corner cases for boxed primitive types and Strings, which also made GUTInfer logic easier. Then we successfully applied GUTInfer and inferred solutions for projects up to 31kSloc.

Finally, we concluded existing problems and interesting future works that are worthy to work on, for example, combing PICO and GUT into a new type system to achieve even finer grained control over mutations and side-effects. We are confident that we'll achieve all these goals in the future.

References

[1] A-normal form. https://en.wikipedia.org/wiki/A-normal_form. Accessed: 2018-04-20.

[2] Automatic type refinement. https://checkerframework.org/manual/#type-refinement. Accessed: 2018-04-21.

[3] Checker Framework. https://github.com/typetools/checker-framework. Accessed: 2018-03-23.

[4] Checker Framework Inference. https://github.com/typetools/checker-framework-inference. Accessed: 2018-03-23.

[5] Google core libraries for Java. https://github.com/google/guava. Accessed: 2018-03-24.

[6] How to create a new checker. https://checkerframework.org/manual/#creating-a-checker. Accessed: 2018-03-23.

[7] Immutable Objects. https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html. Accessed: 2018-03-23.

[8] Immutable, Type declaration annotated with @Immutable is not immutable. http://errorprone.info/bugpattern/Immutable/. Accessed: 2018-03-24.

[9] Lingeling, Plingeling and Treengeling. http://fmv.jku.at/lingeling/. Accessed: 2018-03-23.

[10] LogicBlox. http://www.logicblox.com/technology/. Accessed: 2018-03-23.

[11] Nullness Checker. https://checkerframework.org/manual/#nullness-checker. Accessed: 2018-04-21.

[12] Sat4j, the boolean satisfaction and optimization library in Java. http://www.sat4j.org/. Accessed: 2018-03-23.

[13] Type Annotations and Pluggable Type Systems. https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. Accessed: 2018-03-23.

[14] Uniform access principle. https://en.wikipedia.org/wiki/Uniform_access_principle. Accessed: 2018-04-14.

[15] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*, chapter 97. CRC Press, second edition, Feburary 2004.

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

[17] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 736–747, New York, NY, USA, 2016. ACM.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] W. Dietl, M. D. Ernst, and P. M"uller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, July.

[20] 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, pages 879–896, New York, NY, USA, 2012. ACM.

[21] Jianchu Li. A General Pluggable Type Inference Framework and its use for Data-flow Analysis. available online. https://uwspace.uwaterloo.ca/bitstream/handle/10012/11771/Li_Jianchu.pdf?sequence=1; accessed 2018-03-23.

[22] Coblenz Michael, Sunshine Joshua, and Weber Sam. Exploring Language Support for Immutability. In *IEEE International Conference on Software Engineering*, 2016.

[23] Alexander J. Summers and Peter Mueller. 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, pages 1013–1032, New York, NY, USA, 2011. ACM.

[24] 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, pages 211–230, New York, NY, USA, 2005. ACM.

[25] Dietl Werner, Drossopoulou Sophia, and Muller Peter. Generic Universe Types. In *European Conference on Object-Oriented Programming*, 2007.