UniFlow: A CFG-Based Framework for Pluggable Type Checking and Type Inference

by

Zhiping Cai

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

Waterloo, Ontario, Canada, 2023

© Zhiping Cai 2023

Author's Declaration

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

A type system is a crucial component of high-level programming languages, as it enhances program correctness by ruling out certain type errors. However, the built-in type system often adheres to a specific set of rules defined by the language's specification (e.g., Java, Kotlin and C++). Pluggable type systems were then introduced as an idea to provide customizable type rules for different scenarios.

Various approaches exist for implementing a pluggable type system. The Checker Framework is a well-known framework to facilitate the development of type checkers for Java. This framework enables developers to define their type rules and override the analysis logic. Additionally, Checker Framework Inference is a framework built upon the Checker Framework to provide constraint-based whole-program inference. It helps to reduce the burden of manually annotating the codebase when applying a new type system.

However, the complexity of these frameworks presents a steep learning curve to type system developers. This work examines some of the critical issues encountered from our previous experience in developing these frameworks. The Checker Framework performs its analysis on two different program representations: abstract syntax tree (AST) and control flow graph (CFG). The shared responsibilities of these representations in the framework cause readability and maintainability issues for developers. Checker Framework Inference suffers not only from the same problem but also from difficulty in employing the same type rules for type checking and type inference. This is because the underlying Checker Framework assumes type rules can be checked modularly and immediately at any AST. In contrast, the type inference is not modular and generates constraints to be solved in a later stage.

We propose a novel CFG-based type system framework, UniFlow, addressing the aforementioned issues by providing a unified development process for type systems supporting both type checking and type inference. It strives to resolve types and apply type rules on the program's CFGs whenever possible. This approach reduces friction in type system development, allowing developers to focus on a single flow-sensitive program representation that is simpler than ASTs. It also forces developers to express type rules as constraints, such that the same set of type rules can be implemented once, but consistently reused in type checking and type inference. Moreover, our framework supports running multiple type systems and attempts to improve error message reporting for users.

We present UniFlow's architecture and explain each crucial component and functionality in detail. We discuss the advantages and limitations of our framework. Furthermore, we explore the initial implementation of the framework and outline future research directions.

Acknowledgements

I would like to express my most sincere appreciation to Professor Werner Dietl for his invaluable instruction and support throughout my master's program. His expertise and professional feedback have been extremely helpful in achieving our research goals.

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

I would also like to thank my colleagues Di, Piyush, Haifeng and Alex, for the inspiring discussions and memorable research experience in my graduate journey. I am also grateful for the emotional support from Skittles and Rara.

Dedication

This thesis is dedicated to my parents. Their guidance and encouragement always give me confidence in overcoming any difficulties. I feel grateful for their selfless support.

Table of Contents

List of Figures **ix**

1 Introduction **1**

2 Background **4**

2.1 Program Representations 4

2.2 Pluggable Type Systems 5

2.2.1 Dataflow Framework 5

2.2.2 Checker Framework 9

2.2.3 Nullness Checker 10

2.3 Type Inference 12

2.3.1 Checker Framework Inference 13

3 Introduction to a Novel Pluggable Type System Framework **17**

3.1 Motivations 17

3.1.1 Type Resolution and Dataflow Analysis 17

3.1.2 Duplication of Work 20

3.1.3 Pluggable Types Modeling 21

3.1.4 Type Checking and Type Inference 22

3.1.5 Effective Error Messages 23

3.2 Design Goals 24

4 Architecture and Design **26**

4.1 Architecture Overview . . . 26
4.2 Example and Walk-Through . . . 29
4.3 Data Classes . . . 34
4.4 Framework Layer . . . 36
4.4.1 Analysis Order . . . 37
4.4.2 Slot Location . . . 39
4.4.3 Multiple Type Systems Support . . . 40
4.4.4 Analysis Messages . . . 44
4.5 Type System Layer . . . 45
4.5.1 A Java-Like Programming Language . . . 46
4.5.2 Type Resolution Stage . . . 47
4.5.3 Constraint Generation Stage . . . 53

5 Discussion and Future Work **58**

5.1 Comparison . . . 58
5.1.1 Functionality and Performance . . . 58
5.1.2 Developer and User Experience . . . 60
5.2 Limitations of the Framework . . . 61
5.3 Implementation and Challenges . . . 62
5.4 Future Work . . . 63
5.4.1 Generic Types Support . . . 63
5.4.2 Local Type Inference . . . 64
5.4.3 Bytecode Support . . . 65

6 Related Work **66**

6.1 Type System vs. Flow Analysis . . . 66
6.2 Type Analysis Frameworks . . . 67
6.3 Pluggable Type System Approaches . . . 68

7 Conclusion **69**

References **71**

List of Figures

2.1 An example to demonstrate type checking and type refinement 8
2.2 Overview of the Checker Framework's workflow 8
2.3 Partial type hierarchy of Nullness Checker 11
2.4 Overview of Checker Framework Inference's workflow 14

3.1 An example for explaining type resolution 19
3.2 CFG nodes for the string concatenation in Figure 3.1 20
3.3 An example of compiler error message 23

4.1 Architecture diagram of UniFlow 27
4.2 An example class `Demo` to demonstrate UniFlow's analysis 30
4.3 CFG generated for `Demo::foo` in Figure 4.2 31
4.4 Syntax of our programming language 46
4.5 Relevant CFG nodes to our programming language 47
4.6 Abstract transfer rules for the considered CFG nodes 50
4.7 Abstract type resolution for the considered CFG nodes 52
4.8 Abstract constraint generation for the considered declarations 54
4.9 Abstract constraint generation for the considered CFG nodes 56

Chapter 1

Introduction

A high-level programming language often includes a type system that helps to protect the correctness and safety of a program. The type system defines a set of type rules to verify program behaviour at either compile-time, run-time, or both. For example, Java has a built-in type system that performs checks both statically and dynamically [13], while Python only provides a dynamic type system [33].

A drawback of the built-in type system is it may not satisfy the need of every possible project. Gilad Bracha discussed this limitation and proposed the general idea of a "pluggable type system" in 2004 [6]. He suggested developing type systems that can be used as optional plugins with no impact on the run-time semantics of any programs [6]. This approach allows users to choose the type systems they want for tackling issues not considered by the built-in type system. In this work, we focus on static pluggable type systems for statically-typed programming languages.

Although different type systems analyze different program properties, they all share some basic requirements. They need to define their type lattices, be able to determine the type of a program expression, and properly implement their type rules. To standardize and facilitate the development process, type system frameworks are emerging to provide general architecture and foundation for customizing type systems [3, 12, 27].

The Checker Framework is a framework for building type systems that can perform modular type checking for Java programs [10, 27]. By traversing the abstract syntax trees (ASTs) of a program, it allows type systems to deduce the types of different expressions and check whether there are any type rule violations. It also incorporates flow-sensitive type refinement to reduce the occurrence of false positives, which is achieved by dataflow

analyses on the program's control flow graphs (CFGs). Section 2.2 explains the framework in more detail.

There is also Checker Framework Inference as an extension to the Checker Framework [12, 34]. In addition to the typical type checking, it features a constraint-based whole-program type inference that helps to minimize the burden of manually annotating the source code. If a program passes the type inference, there exists at least one way to annotate the program to make it type check; otherwise, it may have a real type error or false positive result. Since this framework inherits the foundation of the Checker Framework, it has a similar workflow operating on the program's ASTs and CFGs. It introduces constraint variables to applicable locations in the ASTs, and it uses dataflow analysis to refine the variables of some expressions. After that, it traverses the ASTs to generate constraints for type rules. More information about Checker Framework Inference is available in Section 2.3.

From the past experience in developing these two frameworks, we have learned their good practices and areas of improvement. We like the idea of flow-sensitive type refinement, but the implementation is complex in that it requires us to combine the flow-insensitive logic with the flow-sensitive logic. Specifically, to evaluate the type of an expression, the code often needs to interact with the related ASTs and CFGs, leading to complex dependencies and duplicate logic. We also like the idea of expressing type rules as constraints in Checker Framework Inference, but the internal implementation has extra complexity due to its dependency on the Checker Framework. The Checker Framework assumes each type rule can be checked at the location it is applied, but type inference can only generate constraints that need to be solved later. Hence, the framework itself needs to consider two ways to enforce the same type rule: one for type checking, and the other one for type inference.

We propose our framework UniFlow, which unifies the architecture for type checking and type inference, as a novel approach to address the above issues. Inspired by several previous studies on the equivalence between type system and dataflow analysis [15, 19, 26], we decided to strive for a unified program representation and a unified type rule representation. UniFlow resolves types and generates constraints on CFG when possible, thus minimizing the interaction between different program representations. It also ensures type rules are always guarded by constraints, regardless of the current analysis being type checking or type inference. Developers can expect the same set of type rules to have the same behaviour in all situations.

Our framework also attempts to handle other weaknesses in the existing frameworks. For example, Checker Framework Inference does not support type inference on multiple

type systems at the same time, whereas UniFlow supports executing multiple type systems that have no circular dependencies. This functionality is required by some sophisticated type systems, such as the Nullness Checker in the Checker Framework, which depends on an Initialization Checker and a Map Key Checker [32].

The major contributions of this thesis are:

1. An exploration of our learning from the existing frameworks in detail. The experience is helpful for understanding the complexity of a type system framework, as well as the potential areas of improvement.

2. The proposal of a CFG-based pluggable type system framework, UniFlow, that unifies the way for type systems to support both type checking and type inference. We believe there is no precedented framework with a similar design. We present the overall architecture of UniFlow and thoroughly explain its important components and functionalities.

3. We share our current progress in implementing the new framework. There are also some discussions about the advantages, the limitations, and the potential objectives for the future.

The rest of this thesis is organized as follows. Chapter 2 introduces the background knowledge about program representations and type system frameworks. Chapter 3 explores the critical issues according to our past experience and discusses our design goals. Chapter 4 provides the architecture and design details for our framework. Chapter 5 discusses meaningful topics and future directions. Chapter 6 contains related work, and Chapter 7 concludes the thesis.

Chapter 2

Background

2.1 Program Representations

Static program analysis evaluates the properties of a program by analyzing the source code instead of executing it. To avoid directly analyzing the high-level source code, most static analyzers will first translate the source code into an abstract intermediate representation, which is then used as the foundation for analysis [23]. This section introduces the two most common and relevant program representations.

Abstract Syntax Tree

An abstract syntax tree (AST) is a representation of the syntactic structure of a program. It is a tree with nodes representing nested syntax or constructs from the source code, such as an assignment node or a while loop node. AST is essential for analyzing different components of the source code in an organized manner.

Control Flow Graph

A control flow graph (CFG) is a representation of the possible control flows of a program. It usually consists of blocks of nodes and directed edges to connect the blocks, where each node is some program statement and each edge is some flow of control. Compared to AST, there are the following notable differences:

- While a CFG can be generated from an AST [1] or a piece of bytecode [21, 36], it is not possible to convert a CFG back to the original AST. This is because a CFG cannot represent some static constructs, such as a field or a class declaration, that can be expressed by an AST. In addition, multiple kinds of AST may correspond a branching in the CFG, such as an if statement or a ternary expression.
- A CFG models the control flow of a program explicitly, allowing us to easily track the property of an expression at each point of execution. An AST focuses more on the syntactic structure of the source code, so it is less convenient to track the order of execution. Therefore, it is more suitable to perform flow-sensitive analyses on a CFG.
- An AST can be a nested structure with subtrees to represent a complex semantics meaning, such as an entire while loop. A CFG is a flattened structure of blocks and edges, with possibly no syntactic sugars [1]. Figure 2.1 is an example showing the structures of an AST and its corresponding CFG.
- A CFG may introduce temporary local variables that are not in the original AST [1].

Hereinafter, we refer to an AST node as a "tree" and a CFG node as a "node" to avoid any possible confusion.

2.2 Pluggable Type Systems

Creating a pluggable type system is a flexible approach to enforcing custom type rules on program properties that are not limited by the domain of the built-in type system. Type system developers may choose their favourite methods to perform type analysis, while type system users have the freedom to apply the ones they need.

In recent years, there have been many research projects aiming to create new pluggable type systems. Both traditional and novel static analysis methods are being explored, e.g., abstract interpretation [24], dataflow analysis [4, 12, 21], formal verification [8, 18], and machine learning [16, 28].

2.2.1 Dataflow Framework

Dataflow analysis is a way of analyzing certain properties of a program by using a CFG and a lattice [23]. According to Møller and Schwartzbach [23], the analysis will iterate through

all possible execution paths of the program, define constraint variables and constraints for each node, and finally compute the result using a fixed-point algorithm. Based on the order of iteration (i.e., the order of reasoning), there are two different approaches: forward analysis and backward analysis. Forward analysis starts at the entry point of a CFG to collect information and move toward the exit point, whereas backward analysis starts at the exit point of a CFG to collect information and move toward the entry point. In this thesis, we focus on the forward analysis, for a reason that will be explained in Section 2.2.2.

Dataflow Framework is a framework that provides the foundation work for performing dataflow analysis in Java. As written in its documentation:

> The primary purpose of the Dataflow Framework is to estimate values: for each line of source code, it determines properties for each variable, that are true for every value the variable might contain [1].

The framework defines the basic components for dataflow analysis, transforms an AST into a CFG, and provides a fixed-point algorithm.

Basic Components

`Node`, `Block`, and `Edge` together form a `ControlFlowGraph` (CFG). A `Node` represents a single operation in the control flow. A `Block` can be specified as one of the various types of basic blocks: a `RegularBlock` is a sequence of `Node`s that will not raise any exceptions, an `ExceptionBlock` is a single `Node` which may raise an exception, a `ConditionalBlock` represents a branching of the control flow, and finally, a `SpecialBlock` is used as a program entry or exit block. `Edge`s are directed edges connecting different blocks, with labels specifying how the analysis information should flow from one block to another.

A `Store` is a storage for information obtained from previous `Node`s. The two major components of a `Store` are `JavaExpression` and `AbstractValue`. `JavaExpression` is an expression in code that is crucial for the analysis, such as a variable access or a method call. `AbstractValue` is the dataflow information that can be associated with a `JavaExpression`. Therefore, we can view each `Store` as "a mapping from `JavaExpressions` to `AbstractValues` [1]."

A `TransferFunction` is the core of dataflow analysis logic. It is responsible for analyzing an input `Node` and producing useful information. The inputs of a `TransferFunction` are a `Node` and a `TransferInput`, where the `TransferInput` contains one or multiple `Store`s passed from the previous `Node`. The output of a `TransferFunction` is `TransferResult`,

which contains an `AbstractValue` for the current `Node` and one or multiple `Store`s with possibly refined information. The reason for having multiple `Store`s in an input or output is to handle diverged facts at branching (e.g., conditions and if statements).

For brevity, we may use the uncapitalized names interchangeably when referring to instances of these classes (e.g., "transfer function", "transfer input", etc.) after this section.

AST to CFG Transformation

Figure 2.1a shows the source code of an example class `Counter`, and in Figure 2.1b, we have the transformed CFG for the method `countInteger`. We can find different blocks in different shapes: regular blocks as rectangles, conditional blocks as octagons, and exception blocks and special blocks as ovals. Each node is displayed in a separate line in regular blocks, with its expression on the left and its name on the right.

During the transformation, the framework will attempt to desugar each source code expression into simpler node expressions. In the example, we have a postfix increment at line 9 that is desugared in the CFG: a new temporary variable `tempPostfix#num0` is introduced to store the original value of `count`, then the value of `count` is increased by one. More details about the desugaring strategy are available in the framework's manual [1].

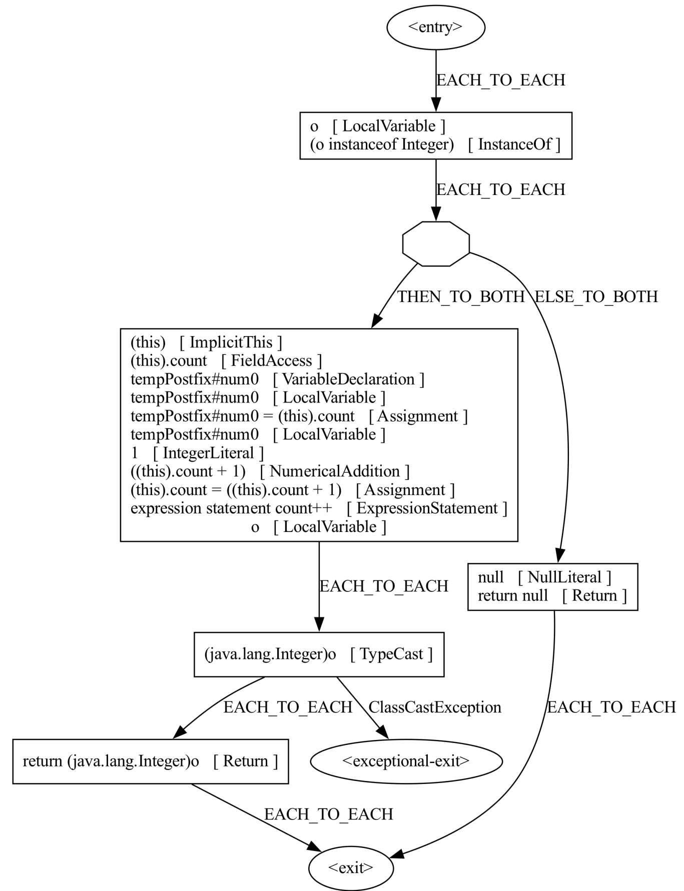
In addition, each CFG node will have a reference to the corresponding AST. For example, the type cast node `(java.lang.Integer)o` has a reference to the return expression tree on line 10 of the source code. If the CFG node is generated from the desugaring process and does not have a corresponding tree, the framework will create an artificial tree to provide as much information as possible [1]. For example, it creates an artificial binary tree for the numerical addition node `((this).count + 1)`.

Analysis and Fixed-Point Iteration

The framework supports both forward analysis and backward analysis. At each node, it prepares a transfer input by merging information from predecessor nodes if this is a forward analysis, or from successor nodes if this is a backward analysis. It then invokes the transfer function with the input and determines if a fixed-point is reached by comparing the returned transfer result with the previous result. The entire process is implemented using a work-list algorithm, in which the work-list keeps track of the blocks that have not reached a fixed-point.

```
1  public class Counter {
2
3    private int count = 0;
4
5    public Integer countInteger(
6      @Nullable Object o
7    ) {
8      if (o instanceof Integer) {
9        count++;
10       return (Integer) o;
11     }
12     return null;
13   }
14 }
```

(a) Source code of an example class `Counter`. The method `countInteger` accepts a nullable Object `o`. If `o` is an Integer, it increases the field `count` by one and returns the Integer; otherwise, it returns null.



(b) CFG generated for `countInteger`

Figure 2.1: An example to demonstrate type checking and type refinement

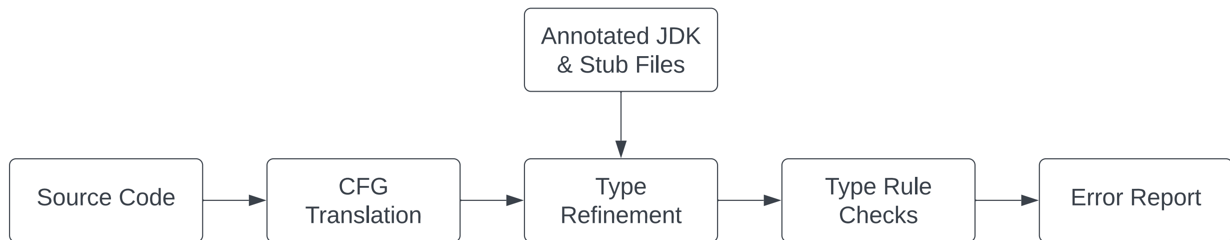


Figure 2.2: Overview of the Checker Framework's workflow

2.2.2 Checker Framework

The Checker Framework provides a software foundation for writing custom pluggable type systems to extend Java's built-in type system. Pluggable types are defined and represented as annotations that can annotate the Java types in the source code. As an example from Figure 2.1a, `Object o` is annotated with a pluggable type qualifier `@Nullable` at line 6. In this thesis, both "annotation" and "qualifier" are used interchangeably to describe a property defined by a pluggable type system.

Many type systems focusing on different program properties have been built on top of the Checker Framework to enhance the code quality of Java programs. For example, there is the Nullness Checker to ensure null safety [27], the PICO Checker to protect immutable properties [30, 31], and the Crypto Checker to detect invalid use of cryptographic algorithms [35].

Figure 2.2 provides a simple overview of the Checker Framework's workflow, in which we highlight the following four processes:

- **CFG Translation:** Dataflow Framework is used to translate ASTs into CFGs. As stated in Section 2.2.1, each CFG node has a reference to the corresponding tree. If the tree is artificial, it will not contain location information for a type system to report relevant errors. Thus, the path from the compilation unit to each artificial tree is recorded so that the innermost real tree can be used for error reporting.
- **Type Refinement:** The Checker Framework attempts to resolve and refine the type of each program expression, by performing dataflow analysis on the generated CFGs. This will be introduced in more detail later in this section.
- **Type Rule Checks:** The Checker Framework traverses different subtrees in each compilation unit and check against the type rules defined by a specific type system. It will report an error message for a tree if the tree fails to satisfy the type rules.
- **Annotated JDK and Stub Files:** Annotated JDK[1] is a fork of the JDK source code with extra annotations from the Checker Framework. The framework can parse these files to retrieve precise types for the most commonly used libraries, and thus reduce the number of false positive outputs. Similarly, developers and users can write stub files to annotate other bytecode or third-party libraries to improve accuracy and precision.

[1] https://github.com/eisop/jdk

The above description shows the Checker Framework heavily depends on Dataflow Framework. This is because Dataflow Framework is an independent module developed along with the Checker Framework, and it is designed to work for general dataflow analysis. More information about the architecture and each component of the Checker Framework is available in its manual [32]. In Section 3.1, we will investigate some details of the type resolving, type checking, and error reporting processes.

Flow-Sensitive Type Refinement

Flow-sensitive type refinement (hereinafter referred to as "type refinement") means the type of an expression can be more precise than its declared type, by analyzing the information from dataflow. Type refinement is an optional feature to type systems since it does not affect the expressiveness of code and requires extra computation resources.

The native type system in javac compiler[2] does not support any type refinements. For instance, in Figure 2.1a, we have an `instanceof` check at line 8 to ensure the Java type of the variable `o` can be refined to `Integer`. However, javac does not recognize this extra piece of information from dataflow and requires an explicit cast at line 10. In fact, it has become a common practice to make an explicit type cast when `instanceof` evaluates to true[3].

The Checker Framework, on the other hand, integrates type refinement to reduce the occurrence of false positives. Since the types are refined as the program execution proceeds, the framework uses forward dataflow analysis to obtain and compute the latest refined type for every expression. Reducing false positives is important to the framework because it makes the analysis result more accurate and trustworthy to users. Without type refinement, for example, Nullness Checker may produce false positive warnings on expressions that are already guarded by a null check.

2.2.3 Nullness Checker

Nullness Checker is one of the earliest built-in checkers in the Checker Framework [27]. It is a sound type system that guarantees no `NullPointerException` will be thrown at runtime, if it type checks the program without reporting errors.

[2] https://openjdk.org/groups/compiler/

[3] Starting from Java 14, a syntactic sugar called "binding variable" [13] is introduced to further simplify this pattern.

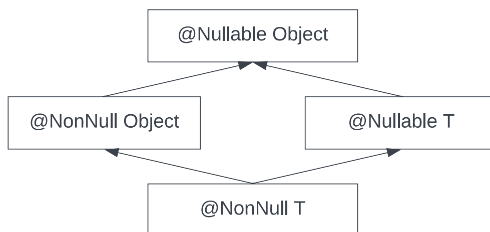


Figure 2.3: Partial type hierarchy of Nullness Checker

The core type qualifiers of Nullness Checker are `@Nullable` and `@NonNull`, where `@Nullable` means the value of a reference can be null and `@NonNull` indicates the value of a reference is definitely not null. The qualifier hierarchy (subtype relationship between qualifiers) of the two is

$$\texttt{@NonNull} <: \texttt{@Nullable}$$

since `@NonNull` is a more precise type than `@Nullable`. When running as a plugin of the javac compiler, the type hierarchy of Nullness Checker is the combination of its qualifier hierarchy and Java's type hierarchy [32]. By generalizing the example provided in the Checker Framework manual [32], this relationship is illustrated in Figure 2.3, where `T` can be any reference type that extends `java.lang.Object`.

In most cases, `@NonNull` is the default qualifier for unannotated type use in the source code. For the example given in Figure 2.1a, the return type of `countInteger` is considered as `@NonNull Integer`, and thus the checker will report an "incompatible type" error at line 12. This error will not appear at line 10 because the type of variable `o` is refined to `@NonNull Object` when the `instanceof` check evaluates to true.

Nullness Checker also contains many other annotations to improve the analysis accuracy. Here, we introduce a few of them:

- `@PolyNull` is a polymorphic qualifier that represents either `@Nullable` or `@NonNull`. It is usually used in a method signature to indicate the method supports both types, and a concrete type will be determined at the call site.
- `@RequiresNonNull` is a precondition to annotate on a method declaration. It means the specified expression should be non-null immediately before the method's call site.
- `@EnsuresNonNull` is a postcondition to annotate on a method declaration. It means the specified expression should be non-null immediately after the method returns.

In addition, Nullness Checker depends on the results from two other checkers: Initialization Checker and Map Key Checker. Initialization Checker can verify different stages of an object's initialization, which helps to ensure all `@NonNull` fields will be initialized with a non-null value. May Key Checker keeps track of the keys of a map to resolve the type of `Map.get` more precisely.

2.3 Type Inference

Type inference, in general, is to statically determine the types of certain variables or expressions that are missing explicit type annotations. There are different kinds of type inference depending on the scope it covers. Java's type system supports various kinds of local type inference, such as "invocation type inference" and "functional interface parameterization inference" [13]. The scope of these type inference techniques is only limited to the local expressions in a method.

When users want to apply a new pluggable type checker into an unannotated code base, a very common issue is the checker will generate too many false positives that make the error report unreadable. This is because the default types are often too conservative to describe many type uses accurately. To improve analysis accuracy and precision, users will need to manually annotate their source code, which requires a certain amount of effort depending on the size of the project. Type inference can help to annotate the code automatically, but inference within a method is not sufficient for determining the types in field or method declarations.

In this thesis, we are focusing on a technique named whole-program inference to reduce the barrier of using a pluggable type system. Whole-program inference determines the type annotation for all possible locations in the source code, by taking the entire program into consideration. For instance, for the source code in Figure 2.1a, we can annotate the locations applicable to whole-program type inference with @i where i is an integer:

```
1 @1
2 public class Counter extends @2 Object {
3
4     public @3 Counter() {
5         super();
6     }
7
8     private @4 int count = 0;
```

```
9
10      public @5 Integer countInteger(
11          @Nullable Object o
12      ) {
13          if (o instanceof @6 Integer) {
14              count++;
15              return (@7 Integer) o;
16          }
17          return null;
18      }
19  }
```

There are several possible solutions to satisfy Nullness Checker's type rules. One of them is as follows:

$$@i = \begin{cases} \texttt{@Nullable} & \text{if } i = 2 \text{ or } i = 5 \\ \texttt{@NonNull} & \text{otherwise} \end{cases}$$

Note that it is possible that some solutions are not favourable or no solutions can be found. The results can still be investigated to show potential issues in the source code, and they always help users work in the right direction toward a safely annotated code base.

For conciseness, we will refer to whole-program type inference as "type inference", unless otherwise specified.

2.3.1 Checker Framework Inference

Checker Framework Inference extends the Checker Framework with the functionality to perform constraint-based type inference. Each type system within the framework supports three modes: a type checking mode for modular type checking, a type inference mode for whole-program type inference, and an annotation mode that annotates the source code with the type inference results [34]. This section explains the type inference mode in more detail.

Figure 2.4 illustrates the workflow of Checker Framework Inference in inference mode. The dashed backward arrow shows the inference result can be used to update the source code, and then the user can execute another round of type check or inference until the result looks satisfying. Compared to the Checker Framework's workflow in Figure 2.2, it has three distinct processes: introducing constraint variables, creating constraints, and solving the constraints.

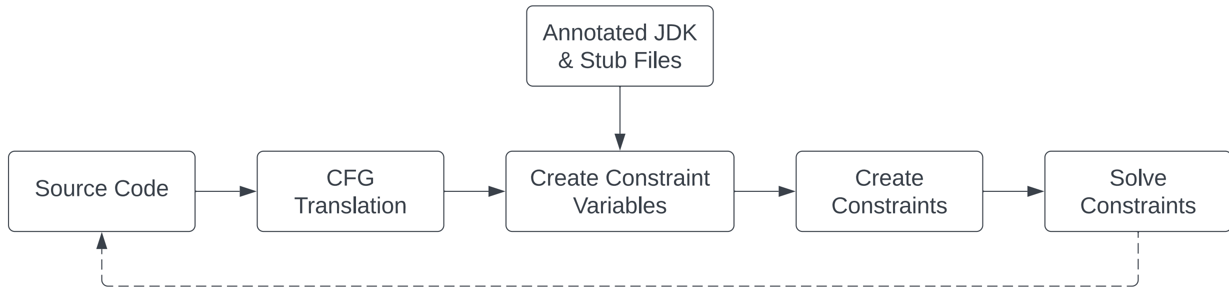


Figure 2.4: Overview of Checker Framework Inference's workflow

Constraint Variable

The creation of constraint variables can be compared to type resolution in type checking mode. They both need to resolve an annotated type for each expression so that the type can be recognized and properly checked by the type rules. However, unlike type checking mode, that can handle the unannotated expressions with defaulting strategy, inference mode has to introduce constraint variables for these locations. The constraint variable to a location can be either fixed or adjustable, based on how the variable is defined.

In Checker Framework Inference, "slot" is the terminology to describe constraint variables. Here, we list a few commonly used slots:

- **Constant Slot** indicates a fixed concrete type that does not require any inference reasoning. This is mainly used for the explicit annotations in the source code, and JDK or other libraries whose types are not involved in the inference.
- **Source Slot** is a fixed variable to an unannotated type use in the source code, including class bounds, field types, method return and parameter types, type parameters, type arguments, etc. Solution to this variable can be inserted back into the source code.
- **Refinement Slot** is created as the refined type of the left-hand side (LHS) expression for each assignment. The purpose of introducing a new Refinement Slot at each assignment is to simulate static single-assignment (SSA) form, thus simplifying the analysis logic. This variable is also fixed to its creation location.
- **Polymorphic Instance Slot** represents a type to substitute polymorphic qualifiers of a method at its call site. Each call site of a poly method requires a new instance of this slot, which will then be fixed for the rest of the inference process.

- **Merge Slot** indicates a least upper bound (LUB) or a greatest lower bound (GLB) of two variables. It is mostly created in dataflow analysis when a join of multiple Stores happens. This slot is not fixed since it can be replaced with a more precise variable as the fixed-point iteration continues.

The first step of constraint variable introduction is to traverse through the input AST and annotate applicable locations with Source Slots. Then the inference proceeds with simulating type refinement for better precision. It will execute a dataflow analysis to create more precise variables, such as Refinement Slots, Polymorphic Instance Slots, and Merge Slots.

Constraint

The commonly used type constraints in Checker Framework Inference are Subtype Constraints and Equality Constraints. The two constraints can be formulated as $x <: y$ and $x = y$, respectively, where x and y are two constraint variables.

There are several places in which constraints can be created. When a variable that associates with some other variables is created, a constraint to describe the relationship among these variables is needed. For instance, a Merge Slot X representing the upper bound of Slot A and Slot B should be a super type of both Slots A and B. The resulting constraint set is $\{A <: X, B <: X\}$

The other possible places are where the Checker Framework applies type rules, such as verifying the well-formedness of a type or applying type rules against an expression. Instead of instantly providing the check results, Checker Framework Inference overrides the associated methods to build constraints when appropriate. Section 3.1.4 gives some discussion about this step.

Constraint Solver

Checker Framework Inference has integrated some commonly used constraint solvers, including MAX-SAT solver, SMT solver, and LogiQL solver. Developers may use one or multiple solvers depending on their preference.

Before feeding the collected constraints into a solver, we need to encode them as the solver's input format (e.g., CNF formula or first-order logic formula). For simple lattices, this process is trivial and may already be supported by the framework. Previous work

from Li has studied how to encode and solve constraints for arbitrary type systems [20]. On the other hand, developers will need to specify how to encode constraints for complex lattices. For example, the PUnits type inference by Xiang et al. [34] defines how to encode constraints as predicates for SMT solver, using boolean and integer variables.

Chapter 3

Introduction to a Novel Pluggable Type System Framework

This thesis introduces UniFlow, a novel approach for developing a pluggable type system framework. Instead of writing analysis code for both the AST and the CFG, the framework allows type system developers to focus on a single program representation (i.e., CFG). Instead of using different approaches in type checking and type inference, the framework bridges the gap with a constraint-based approach for both. To provide a better understanding of the decisions taken, in this chapter, we explain the major motivations and design goals behind the initiation of UniFlow.

3.1 Motivations

3.1.1 Type Resolution and Dataflow Analysis

Type systems need to first resolve some declarations or uses of a type before performing any analyses. Type resolution can be either flow-insensitive or flow-sensitive (i.e., flow-sensitive typing). From Chapter 2, it is intuitive to utilize AST for flow-insensitive analysis and CFG for flow-sensitive analysis. However, evaluating the type of the same expression in both program representations overcomplicates the responsibilities of the two analyses. For a pluggable type system framework with such architecture, the extra complexity is a burden to the framework maintainers and the type system developers.

According to the Checker Framework Manual [32], `CFAbstractTransfer` and `AnnotatedTypeFactory` are two major components for type resolution. The class `CFAbstractTransfer` and its child classes are transfer functions in the dataflow analysis to provide flow-sensitive type refinement. The class `AnnotatedTypeFactory` and its child classes provide the most accurate type information, so they need to combine type resolution from both flow-sensitive and flow-insensitive analyses. During type refinement process, the transfer function unfortunately requires `AnnotatedTypeFactory` to provide some type information. Thus, there is a tight dependency between the flow-sensitive and the flow-insensitive logic, and a developer has to study the source code in order to understand details about the two sides.

Here, we provide a brief overview of the dependency[4]. In `CFAbstractTransfer`, `AnnotatedTypeFactory` is often used for retrieving a flow-insensitive type of a tree associated with the current CFG node, such as the node of a field access, an array access, and a local variable access. If another type of the current node is available in the input store of the transfer function, the more specific type of the two will be determined and returned as the transfer result value. Otherwise, the flow-insensitive type will be used as the default transfer result value.

In `AnnotatedTypeFactory` and its child class `GenericAnnotatedTypeFactory`, we have the foundation of type resolution logic in the Checker Framework. A method called `getAnnotatedType` is the commonly used entry point for type resolution, which has many variants (overloaded and extended versions) to accommodate varying situations. In general, it determines the type of an AST with the following steps:

1. Create an instance of `AnnotatedTypeMirror` as a mirror of the Java type associated with the tree, which can store any annotations resolved by a type system.
2. Annotate the type by migrating the existing annotations from the source code.
3. Annotate the type by traversing and analyzing information from the tree. For example, the type of a binary expression is the LUB of the operands' types.
4. Annotate the type by finding related type information. For example, if a wildcard type has unannotated bounds, propagates the annotations from the substituted type parameter to the bounds.
5. Apply default annotations according to any predefined rules.

[4] Information is collected from version 3.28.0-eisop1 of the Checker Framework at https://github.com/eisop/checker-framework.

```
1  public class TypeResolve<T> {
2
3      List<? extends T> rankings = new ArrayList<>();
4
5      void printTop1() {
6          if (!rankings.isEmpty()) {
7              System.out.println("Top 1 is " + rankings.get(0));
8          }
9      }
10 }
```

Figure 3.1: An example for explaining type resolution

6. Attempt to query and apply flow-sensitive type refinement for the tree (if the tree is not the exact same one being processed by a transfer function).

In steps 3 to 6, the type factory may need to recursively resolve the types of some other components, which exacerbates the complexity and makes debugging a demanding task.

Consider the code snippet in Figure 3.1, the string concatenation in line 7 is a good example for demonstrating type resolution. We highlight a portion of steps that will be performed in the Checker Framework, and we will study how to simplify them in Section 3.2:

- The transfer function receives a CFG node `StringConcatenateNode` which represents the binary tree "`Top 1 is " + rankings.get(0)`, then it will ask `AnnotatedTypeFactory` for the type of the tree.
- Following the previously described steps for `getAnnotatedType`, at step 3, the type factory needs to recursively invoke `getAnnotatedType` to resolve the types of the left and right operands.
- Finding the type of the left operand is trivial since we only need flow-insensitive information from a string literal.
- Finding the type of the right operand requires a propagation of annotations from the type parameter `T` as described previously in step 4. Flow-sensitive type refinement will also be queried and applied.

Node Expression	Node Type
"Top 1 is "	StringLiteralNode
(this)	ImplicitThisNode
(this).rankings	FieldAccessNode
(this).rankings.get	MethodAccessNode
0	IntegerLiteralNode
(this).rankings.get(0)	MethodInvocationNode
StringConversion((this).rankings.get(0))	StringConversionNode
("Top 1 is " + StringConversion((this).rankings.get(0)))	StringConcatenate

Figure 3.2: CFG nodes for the string concatenation in Figure 3.1

- Once the types of the two operands are ready, the type factory handles any implicit conversions of the operands[5]. In this example, `rankings.get(0)` is implicitly converted to a `String` by Java, so the corresponding Java type and annotations need to be adapted.

- Finally, the LUB of the two adapted types is computed and returned.

3.1.2 Duplication of Work

There are many similar components between an AST and the corresponding CFG. For example, in the Dataflow Framework [1], `AssignmentNode` and `MethodInvocationNode` correspond to `AssignmentTree` and `MethodInvocationTree` respectively. These redundancies also become a burden to type system developers, as the code is prone to redundant computation and duplicate logic.

Taking the string concatenation shown in Figure 3.1 as input, the block of CFG nodes generated is available in Figure 3.2. Note that flow-sensitive analysis has to process the nodes in the (execution) order they are defined inside a block. Recall from the previous section that, to resolve the type of the last node, the Checker Framework needs to compute the types of the two operands. The computation is redundant because, at the time of analyzing the last node, both operands are already analyzed, and thus their types are already available.

[5]There was a recent issue related to this step, where the bounds of an annotation could be different between pre- and post-conversion. The issue was causing invalid annotations in both flow-sensitive and flow-insensitive type resolutions. Finding and fixing the issue was a demanding and time-consuming task, as shown in the details provided at https://github.com/eisop/checker-framework/pull/213.

Desugaring is another area prone to duplicate work. Since many syntactic sugars are defined in high-level programming languages, type systems are required to desugar the code to handle any implicit operations. When a type system requires both AST and CFG, the amount of effort is naturally doubled. We can examine the string concatenation example again because there is an implicit string conversion for its right operand. The Dataflow Framework has built-in functionality for desugaring [1], which generates the `StringConversionNode` shown in Figure 3.2. However, the `getAnnotatedType` method must still handle the implicit conversion in a flow-insensitive context, as described in the previous section.

3.1.3 Pluggable Types Modeling

During type resolution, a type system will deduce the correct types and store the result as a well-formed data object. This section discusses the issues we have encountered when using such data objects in the Checker Framework.

In the Java compiler, an instance of `TypeMirror` or its child classes represents a use of some Java types and the corresponding details. In the Checker Framework, `AnnotatedTypeMirror` closely follows the structure of `TypeMirror` to not only provide the underlying Java type but also incorporate annotations for any pluggable types.

The first issue is about the mutability of data objects. `AnnotatedTypeMirror` provides various setter methods to modify its internal data about Java types and annotations. Although it gives the flexibility to adjust the data for use cases like type refinement, developers need to be cautious that changing one instance of `AnnotatedTypeMirror` will affect all references to it. To reuse an `AnnotatedTypeMirror` without affecting other references, utility methods `shallowCopy` and `deepCopy` are created for different levels of adjustment. Developers may question why a copy method is used and how the copy may affect other places in code. Without clear documentation, it is difficult to answer those questions as the code base continues to grow larger. Pull requests #1086, #1622, #2176 in the typetools Checker Framework[6] are examples with such discussions, in which we see questions like "Why is a shallow copy good enough?" and "Why is a deep copy necessary?" and an answer could be "I changed it to a deep copy, but I'm not sure if there is other code that relies on the shallow copy."

The second issue is how to distinguish between the declaration and the use of a type. For example, in the following code snippet:

[6] https://github.com/typetools/checker-framework

```
1 class Box<T> {
2     Box<T> nestedBox;
3 }
```

`Box<T>` in line 1 is the declarations of type `Box` and its type parameter `T`, while `Box<T>` in line 2 is a use of `Box` with type argument `T`.

In the Java compiler, `Element` represents a static construct that includes all kinds of type declarations, which is different from a `TypeMirror` (i.e., a type use). In the Checker Framework, `AnnotatedTypeMirror` can be adapted to represent either an annotated declared type or an annotated type use. It introduces a helper method `isDeclaration` for developers to distinguish between the two. Consider the two possible cases of an `AnnotatedTypeMirror` representing `@X Box<T>`:

- If it is the declaration of `Box`, all uses of `Box` must not have an annotation `@Y` under the same pluggable type hierarchy such that `@X` <: `@Y` and `@X` ≠ `@Y`[7]. In addition, there is a method named `getTypeArguments` which actually returns the type parameter `T`.
- If it is a use of `Box`, `@X` is simply the primary annotation on the type use, and `getTypeArguments` intuitively returns a type argument `T`.

The problem is the meaning of an `AnnotatedTypeMirror` varies based on the result of `isDeclaration`, and developers may misuse one as the other.

3.1.4 Type Checking and Type Inference

Checker Framework Inference, which is built upon the foundation of the Checker Framework, demonstrates that the same type rules can be reused in both type checking and type inference [12]. While the ideal implementation should also reuse most of the architectures and code, Checker Framework Inference maintains two sets of implementation for different modes: a traditional checker for type checking mode and a constraint-based checker for type inference mode. The main reason is it inevitably inherits the architectures and interface designs that solely target type checking from the Checker Framework.

In type checking, the Checker Framework needs to compute some type relations according to the tree it is processing. For instance, for an `AssignmentTree`, it needs to determine

[7] This does not apply to local variables because their type can be temporarily out of the bound and then refined to a type within the bound. This allows intermediate steps for object initialization.

```
1   Nullness.java:5: error: [dereference.of.nullable] dereference of
        possibly-null reference object
2           object.toString();
3           ^
```

Figure 3.3: An example of compiler error message

whether the type of the RHS is a subtype of the type of the LHS. The result of each type relation check is a Boolean value because all the involved types can be determined when visiting the tree.

On the other hand, instead of immediately checking some type relations, type inference requires collecting and encoding each type rule check into a set of constraints. Then it utilizes a constraint solver to determine a satisfying solution. Therefore, when overriding some type checking logic, Checker Framework Inference always needs to create new constraints on the side while returning a meaningless Boolean value. Developers will unfortunately have to take care of the consequences of returning such a meaningless result. If the result is always true, an underlying disjunction will pass instantly and skip the rest of the checks; if the result is always false, an underlying conjunction will fail and also ignore the remaining checks.

3.1.5 Effective Error Messages

When there are failures in type checking or type inference, the type system should output error messages to help users identify potential issues in the source code. According to Becker et al. [5], previous studies have demonstrated empirical evidence that readability and context information of error messages are crucial to developers' productivity.

The Checker Framework provides a strong foundation for type systems to report their errors. It supports referring to a message template with an identifier key in the source code, while the actual key-value pair of message templates are stored in a separate Java properties file [32]. This is helpful for reviewing the message contents and keeping the source code clean. It also utilizes the built-in error reporting tools from the Java compiler to produce enhanced and consistently styled error messages. In Figure 3.3, the Checker Framework provides the error message "dereference of possibly-null reference object" with message key "dereference.of.nullable", and the Java compiler provides location information that the error happens on the use of variable `object` on line 5 of the file `Nullness.java`.

However, this inherited error reporting architecture is unsuitable for Checker Framework Inference. As described in the previous section, the Checker Framework can obtain the type checking results in real time, so it naturally reports an error if a check fails. Because giving immediate results is impossible in type inference, a type system cannot decide what to report when the Checker Framework asks for error reporting. Currently, Checker Framework Inference can output a minimal unsatisfiable subset of constraints, with bytecode style locations about where the constraints are created. These error messages could be duplicated when multiple unsatisfiable constraints are created for a single check, and their information cannot be easily interpreted by users.

3.2 Design Goals

Taking all the topics discussed in Section 3.1 into consideration, we want our UniFlow framework to provide a better experience for both type system developers and users. Hence, we carefully decide the following design goals for the new framework.

- **Single Program Representation**: Given the source code of a sequence of (execution) statements, UniFlow will convert it into a CFG and provide the foundation to allow type systems to perform type resolution, type checking and type inference on top of the CFG. This removes the extra complexity of learning and using both AST and CFG, which is studied in Section 3.1.1. In addition, since the CFG generator can take care of code desugaring, developers can focus on each straightforward CFG node instead of being concerned about syntactic sugars such as implicit conversion and compound assignment. This simplifies and deduplicates some of the logic described in Section 3.1.2.

  The only exception is inheritance-related logic because inheritance is the property of a static construct that is irrelevant to control flow. Since the type rules for inheritance are mostly straightforward, the framework can provide a general default implementation for most type systems. Developers may also override the default behaviour to fulfill special requirements.

- **Incremental Type Resolution**: When resolving the type of a CFG node, UniFlow should allow developers to retrieve the types determined for some previous nodes, upon which the current type can be built. This addresses the duplicated work of type computation in Section 3.1.2.

- **Multiple Type Systems Support**: UniFlow should support running multiple type systems at the same time, where there can be non-cyclic dependencies between type systems. This is supported in the Checker Framework, but it is not adapted to work in Checker Framework Inference because of some technical challenges.

- **Immutable Data Objects**: Pluggable types and other widely used data objects should be immutable. This allows developers to easily create altered versions of data objects, regardless of the size of the code base or the number of references to an object. They will not be distracted by how to copy an object or potential issues to some irrelevant uses of it. This should substantially mitigate the first issue in Section 3.1.3.

- **Distinct Declaration Constructs**: There should be distinct data classes for static declaration constructs that are meaningful to type systems. In Java, these constructs include type declaration, method declaration, variable declaration, etc. This emphasizes the differences between context-sensitive and context-insensitive logic in the framework, which may help with smoothing the learning curve shown in Section 3.1.1 and 3.1.3. In addition, this can also reduce some duplicated work relevant to issues in Section 3.1.2. For example, type resolution for a declaration does not require any refinements, so it can be computed once for all future references.

- **Constraint-Based Type Rules**: UniFlow will unify the experience of type checking and type inference by abstracting each type rule into a constraint object. In type checking mode, all constraints can be solved in real time, so we can simply use two special constraint objects to represent constant results "true" and "false". Real constraints will be created in type inference mode, and the framework will attempt to solve them in a later stage. This feature not only addresses the issues in Section 3.1.4, but also makes the code reusable for both type checking and type inference modes. Developers can simply provide constraints as type rules for each CFG node, and users are guaranteed to observe a consistent behaviour across both modes.

- **Effective Error Reporting**: UniFlow should provide a strong foundation for error reporting, which produces readable and consistent error messages similar to the example in Figure 3.3. To match the strategy for type rules, developers will be able to reuse the code in both type checking and type inference. Additionally, a dedicated file or storage for message templates should still be supported. By addressing the issues in Section 3.1.5, type system users can actually expect error messages to help them locate and understand the errors.

Chapter 4

Architecture and Design

In this chapter, we present the overall architecture of UniFlow, explain the framework's process with a complete example, and discuss the design of each component in detail. We will demonstrate our work based on the Java programming language, whereas the general ideas can be adapted to other object-oriented or imperative languages as well.

4.1 Architecture Overview

Figure 4.1 shows the overview of UniFlow's architecture. We split most components into two layers, the framework layer and the type system layer, according to their functionality and responsibility. For clarity and simplicity, only the major interactions among components are illustrated using arrows, while the comprehensive relationships will be explained throughout the rest of Chapter 4. The obligation of each component is described as follows:

- **Javac Compiler** is the official Java compiler[8]. During compilation, it generates an AST for each compilation unit, and then it passes the AST to its compiler plugins for additional static analyses[9]. Our framework is also a compiler plugin.
- **CFG Builder** accepts ASTs and translates them into CFGs. We use Dataflow Framework in this translation and the following dataflow analyses.

[8] https://openjdk.org/

[9] More details about compiler plugins are available at https://openjdk.org/groups/compiler/processing-code.html and https://openjdk.org/groups/compiler/doc/hhgtjavac/index.html.

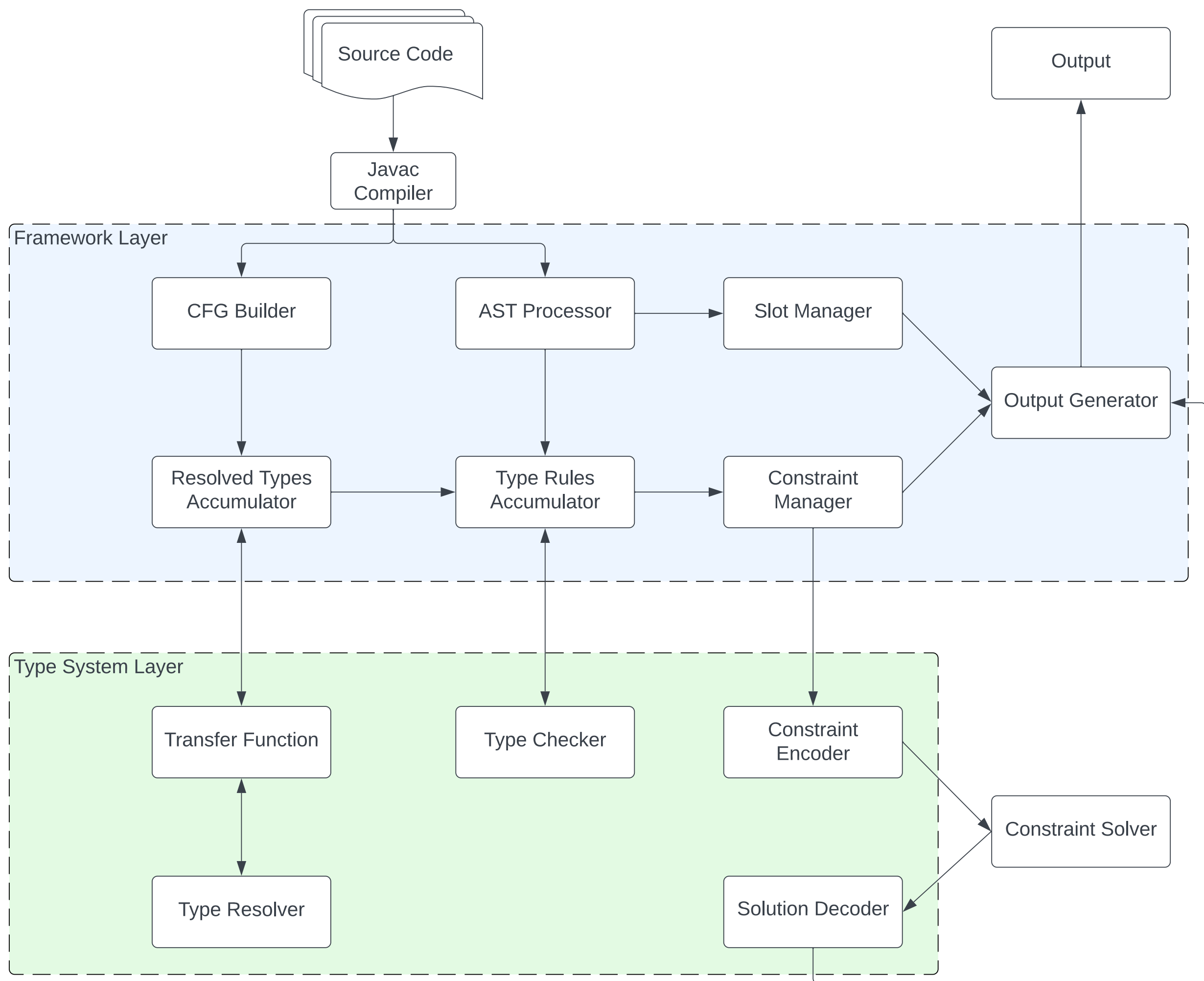


Figure 4.1: Architecture diagram of UniFlow

- **Resolved Types Accumulator** accepts a CFG and starts type resolution in the form of a dataflow analysis. It iterates through each type system and triggers their transfer functions. The fixed-point status for different type systems is memorized for the upcoming type checking stage.

- **Transfer Function** defines how a type system should manage type resolution and type refinement for each node. It invokes the type resolver to determine the most precise type of the current node, updates the stores for type refinement, and constructs a transfer result containing the latest type and stores.

- **Type Resolver** determines the flow-sensitive type of a node or the flow-insensitive type of a declaration construct. Given a node with its transfer input, the resolver returns its qualified type (i.e., fully annotated Java type). Given a declaration construct, the resolver annotates all type declarations and type uses in it.

  In type checking mode, it simply produces concrete types using collected facts and defaulting strategy. Whereas in type inference mode, it may need to introduce new constraint variables. The constraint variables are represented as slots similar to the ones in Checker Framework Inference [12].

- **AST Processor** accepts the AST of a compilation unit and triggers tasks that require trees. This is to cover the information that is missing in a CFG, such as some declaration constructs and the locations of some specific tokens in the source file. In the current design, we find it necessary only for the type rules accumulator and the slot manager.

- **Type Rules Accumulator** iterates through each type system and applies their type checker on the consumed CFG or AST. If the input is a CFG, it provides the type checker with every node in the CFG and the associated fixed-point status, namely, transfer input and transfer result. If the input is an AST, it invokes the type checker to check inheritance rules on class and method declarations.

- **Type Checker** generates constraint-based conditions to enforce the type rules. As mentioned in the item above, it defines rules for any nodes and also contains inheritance checks for class and method trees. Note that the rules and the conditions are mode-agnostic, which guarantees consistent behaviour in both type checking and type inference.

- **Slot Manager** is a centralized place for managing slots. When the type resolver requests a constraint variable, it must validate the request and determine the correct slot to return.

Tracking the locations of different slots is another important task in slot manager. This serves two purposes: one is to provide debug information about the creation of each slot, and the other one is to insert inference results back into the source code. It traverses the AST from the AST processor to memorize all possible locations that can be annotated in the source code.

- **Constraint Manager** is a centralized place for managing constraints. For each type of relation (e.g., subtype, equality), it determines the most accurate constraint to return. If all operands are constant slots, it can return an "always true" or "always false" constraint; otherwise, it will consider creating a real constraint. It also maintains a mapping from a set of constraints to the corresponding error messages created by each type system.

- **Constraint Encoder** accepts the input constraint objects and encodes them into formulas that can be recognized by a specific constraint solver.

- **Constraint Solver** solves the encoded constraints. If a solution exists, it returns the solution; otherwise, it returns the minimal unsatisfiable subset of constraints. Note that there are no actual constraints in type checking mode, so a solver is only necessary for type inference mode.

- **Solution Decoder** decodes the solution from the solver and generates a mapping from each slot to its assigned concrete qualifier.

- **Output Generator** can provide meaningful analysis results to both users and developers, by collecting all information from the slot manager, the constraint manager, and the inference solution if applicable. For type system developers, it generates details about each slot and constraint created. For type system users, its outputs are different based on the analysis result. If inference succeeds, users can expect the solution to be inserted back into the source code. If checking or inference fails, users will see the error messages deduced from those unsatisfiable constraints.

4.2 Example and Walk-Through

In this section, we provide an example program to demonstrate the entire type checking and type inference processes in UniFlow. The example is a nullness analysis based on the type system from Nullness Checker (introduced in Section 2.2.3), without advanced annotations (e.g., pre- and post-conditions) or initialization checks.

```
1  @1
2  class Demo extends @2 Object {
3
4    @3 Demo foo(@4 boolean isNonNull) {
5      @5 Demo result = null;
6      if (isNonNull) {
7        result = new @6 Demo();
8      }
9      return result;
10   }
11 }
```

Figure 4.2: An example class `Demo` to demonstrate UniFlow's analysis

Figure 4.2 shows the example class `Demo`. The `foo` method in `Demo` has a boolean parameter `isNonNull`. If `isNonNull` evaluates to true, the method returns a new instance of `Demo`; otherwise, it simply returns null. The annotations `@1` to `@6` do not appear in the source code, but each indicates a possibly unannotated location and can have a different meaning in different type systems. In this analysis, `@2` and `@6` are meaningless to a nullness type system, so they are fixed to `@2` = `@Nullable` and `@6` = `@NonNull`.

When javac compiler receives the file "Demo.java", it generates an AST for the compilation unit `Demo` and passes the AST to UniFlow. CFG builder receives the AST and uses Dataflow Framework to generate CFGs for every enclosed initializer block, field initializer, constructor, method, and lambda expression. In this example, we are only interested in the `foo` method. Its corresponding CFG is shown in Figure 4.3, in which each non-special block is numbered for future references.

Now, we are at resolved types accumulator for type resolution. The accumulator will invoke the transfer function of our single type system. Since the example CFG does not have any execution loops, the transfer function will terminate after two iterations through the CFG (where the second iteration is to confirm we have reached the fixed-point). The fixed-point results are as follows, where S_i and S'_i are the stores before and after block $\#i$ respectively, and $[\![\texttt{X}]\!]$ represents the resolved type of node `X`.

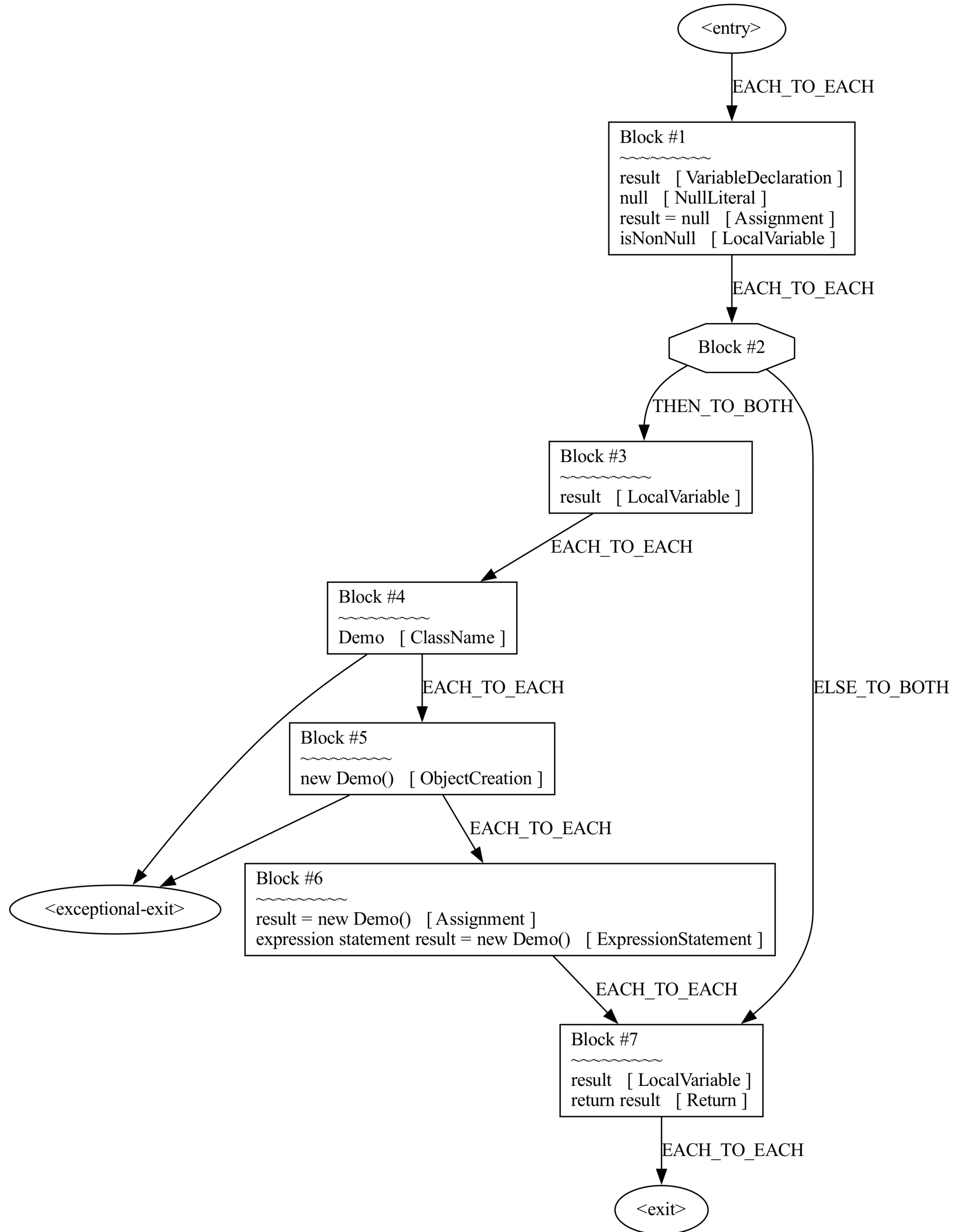


Figure 4.3: CFG generated for `Demo::foo` in Figure 4.2

$$
Block\#1: \begin{cases}
S_1 = \{\texttt{isNonNull} \mapsto \texttt{@4 boolean}\} \\
[\![\texttt{result}]\!] = \texttt{@5 Demo} \\
[\![\texttt{null}]\!] = \texttt{@Nullable NullType} \\
[\![\texttt{result = null}]\!] = \texttt{@7 Demo} \\
[\![\texttt{isNonNull}]\!] = \texttt{@4 boolean} \\
S_1' = S_1 \{\texttt{result} \mapsto \texttt{@7 Demo}\}
\end{cases}
$$

$$
Block\#2: \begin{cases}
S_2 = S_1' \\
S_2' = S_2
\end{cases}
$$

$$
Block\#3: \begin{cases}
S_3 = S_2' \\
[\![\texttt{result}]\!] = \texttt{@7 Demo} \\
S_3' = S_3
\end{cases}
\qquad
Block\#4: \begin{cases}
S_4 = S_3' \\
[\![\texttt{Demo}]\!] = \emptyset \\
S_4' = S_4
\end{cases}
$$

$$
Block\#5: \begin{cases}
S_5 = S_4' \\
[\![\texttt{new Demo()}]\!] = \texttt{@NonNull Demo} \\
S_5' = S_5
\end{cases}
$$

$$
Block\#6: \begin{cases}
S_6 = S_5' \\
[\![\texttt{result = new Demo()}]\!] = \texttt{@8 Demo} \\
[\![\texttt{expression ...}]\!] = \emptyset \\
S_6' = S_6 \{\texttt{result} \mapsto \texttt{@8 Demo}\}
\end{cases}
$$

$$
Block\#7: \begin{cases}
S_7 = \mathrm{LUB}(S_2', S_6') \\
\quad = \{\texttt{isNonNull} \mapsto \texttt{@4 boolean}, \texttt{result} \mapsto \texttt{@9 Demo}\} \\
[\![\texttt{result}]\!] = \texttt{@9 Demo} \\
[\![\texttt{return result}]\!] = [\![\texttt{result}]\!] \\
S_7' = S_7
\end{cases}
$$

At block #1, we create a new constraint variable `@7` as the refined type of `result` after the assignment. We learned this technique for simulating SSA form from Checker Framework Inference[10]. At block #2, we have a "then" store and an "else" store for the branching, but we only show S_2' because the two stores are identical in this case. Equations

[10] https://github.com/eisop/checker-framework-inference/blob/47bf3c36b99bfeb4d86ab69857c40033ea774c7c/src/checkers/inference/model/RefinementVariableSlot.java

from block #3 to block #5 are straightforward. At block #6, we introduce another new constraint variable `@8` for the assignment. At block #7, our input store is the LUB of S'_2 and S'_6 because we need to merge the facts from the two incoming edges. As a result, the constraint variable `@9` is used to represent the LUB of `@7` and `@8`. Note that the types of two nodes, ⟦`Demo`⟧ and ⟦`expression ...`⟧, are ignored because they do not have a Java type. ⟦`Demo`⟧ is a class name identifier, and ⟦`expression ...`⟧ is just an internal marker for some evaluated program statement.

The actual objects for constraint variables `@i` are defined by type resolver, which can be different in different contexts. In type checking mode, there are no real constraint variables because we can always reason a concrete qualifier. We will have `@3` = `@4` = `@6` = `@8` = `@NonNull` and `@1` = `@2` = `@5` = `@7` = `@9` = `@Nullable`. Whereas in inference mode, we need real constraint variables: `@1` to `@6` are source slots, `@7` and `@8` are refinement slots, and `@9` is a merge slot.

The next major task is to generate constraints for type rules. As the type checker iterates through the CFG with cached fixed-point results, we will collect the following constraints.

`@7` <: `@5`	(at ⟦`result = null`⟧)
`@7` = `@Nullable`	(at ⟦`result = null`⟧)
`@8` <: `@5`	(at ⟦`result = new Demo()`⟧)
`@8` = `@NonNull`	(at ⟦`result = new Demo()`⟧)
`@7` <: `@9`	(at block #7)
`@8` <: `@9`	(at block #7)
`@9` <: `@3`	(at ⟦`return result`⟧)

Recall that the type checker must also apply type rules to declaration trees. The generated constraints are

`@2` <: `@Nullable`	(upper bound of `Object` is `@Nullable`)
`@1` <: `@2`	(`Demo` extends `Object`)
`@3` <: `@1`	(upper bound of `Demo` is `@1`)
`@4` <: `@NonNull`	(upper bound of `boolean` is `@NonNull`)

Finally, we want to solve the constraints and report the analysis results. In type checking mode, we can directly solve the constraints by substituting constraint variables with the aforementioned assignments. The type system will find the constraint `@9` <: `@3` is not satisfiable, and thus it will report an "incompatible return type" error for the return

statement. In type inference mode, we can utilize the method from Li's work [20] to encode the constraints as CNF formulas, and then feed the encoded constraints into a MAX-SAT solver. A possible solution is `@4` = `@6` = `@8` = `@NonNull` and `@1` = `@2` = `@3` = `@5` = `@7` = `@9` = `@Nullable`. Inserting the solution back, we will get an altered version of the source code that can pass the type check:

```
1  @Nullable
2  class Demo extends @Nullable Object {
3
4    @Nullable Demo foo(@NonNull boolean isNonNull) {
5      @Nullable Demo result = null;
6      if (isNonNull) {
7        result = new @NonNull Demo();
8      }
9      return result;
10   }
11 }
```

Note that some qualifiers are the same as the default in type checking mode. Therefore, we can further simplify the result by removing the redundant explicit qualifiers:

```
1  class Demo extends Object {
2
3    @Nullable Demo foo(boolean isNonNull) {
4      Demo result = null;
5      if (isNonNull) {
6        result = new Demo();
7      }
8      return result;
9    }
10 }
```

4.3 Data Classes

The word "data class" describes a class whose primary purpose is to hold different data properties. We are using it in this thesis as a more intuitive alternative to other names,

such as "value class". In UniFlow, important data objects are required to be immutable for code simplicity and reusability. We define the following core immutable data classes:

Definition 4.3.1 (`Qualifier`)**.** `Qualifier` is an internal representation of annotation in the framework. For each pluggable type in the lattice, there should be a unique corresponding `Qualifier`.

Definition 4.3.2 (`QualifiedType`)**.** `QualifiedType` mirrors the structure of Java's `TypeMirror`[11] with attached qualifiers. Each `QualifiedType` can have at most one primary qualifier (i.e., the annotation directly annotated on the type).

Definition 4.3.3 (`QualifiedElement`)**.** `QualifiedElement` mirrors the structure of Java's Element[12] with attached qualifiers. Each `QualifiedElement` can have at most one primary qualifier (i.e., the annotation directly annotated on the element).

Definition 4.3.4 (`Slot`)**.** `Slot`s are a special subset of `Qualifier`s to represent constraint variables [12].

Definition 4.3.5 (`Constraint`)**.** `Constraint` represents some abstract relation among a set of `Slot`s [12].

Definition 4.3.6 (`ProductSlot`)**.** `ProductSlot` is a special implementation of `Qualifier`. It represents an n-tuple $(S_1, \ldots, S_n)$ in a product lattice $L_1 \times \cdots \times L_n$ where each S_i is a `Slot` representing a constraint variable in lattice L_i, and $L_i = L_j$ if and only if $i = j$. `ProductSlot` is not a subtype of `Slot`.

Definition 4.3.7 (`AnalysisMessage`)**.** `AnalysisMessage` is identified by a 3-tuple (K, P, M) where K is the kind of the message (e.g., warning or error), P is the position that the message is referring to in the source code, and M is the message key and message body.

`Qualifier`, `QualifiedType` and `QualifiedElement` are the essential components for creating annotated types and annotated declaration constructs, which will be used throughout the analysis. The purpose of introducing `Qualifier` is for flexibility and deduplication. Although Java has an existing annotation representation called `AnnotationMirror`[13], it is

[11] https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/lang/model/type/package-summary.html

[12] https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/lang/model/element/package-summary.html

[13] https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/lang/model/element/AnnotationMirror.html

a simple key-value mapping of the annotation properties and is not flexible for customization. In addition, one instance of internal `Qualifier` may represent multiple annotations that are alias to each other. For example, `@a.b.c.Nullable` and `@x.y.z.Nullable` from different packages can both be recognized as a `NULLABLE` enum instance internally.

In contrast to `AnnotatedTypeMirror` in the Checker Framework, `QualifiedType` is an annotated type use, while `QualifiedElement` is an annotated declaration construct. This helps developers to distinguish the difference between the two and thus will act more carefully when converting from a type declaration to a type use. Note that type uses could appear in a declaration construct, such as the extends bound in a class declaration. This means `QualifiedType`s can be nested inside a `QualifiedElement`, but a `QualifiedType` cannot contain a `QualifiedElement`.

`Slot`, `ProductSlot` and `Constraint` are essential for type rules enforcement. The definitions of `Slot` and `Constraint` are based on the ones defined in Checker Framework Inference [12]. However, having the two components is not sufficient to UniFlow, as we need to support `Slot`s and `Constraint`s generated by different type systems. In Definitions 4.3.2 and 4.3.3, we set one as the upper bound of the number of primary qualifiers in a `QualifiedType` or a `QualifiedElement`. This is to simplify their interfaces and offload the complexity of managing multiple `Slot`s to a specific implementation.

`ProductSlot` is the special implementation to support `Slot`s from multiple lattices and type systems. As described in Definition 4.3.6, each `ProductSlot` can be represented as $(S_1, \ldots, S_n)$, where n is a fixed number based on the total number of lattices involved in the analysis. For example, in an analysis with two distinct type systems T_1 and T_2, where each type system T_i has one type lattice L_i, then all `ProductSlot`s in the analysis have size two. If `@1` $\in L_1$ and `@2` $\in L_2$, we can represent `@1 @2 Object` (multiple primary qualifiers) as `@(@1,@2) Object` internally with a `ProductSlot`. Note that a `ProductSlot` may contain both constant and non-constant slots, which is a combination of constraint variables that are either solved or yet to be solved. In addition, to prevent needlessly nested structures, `ProductSlot` is designed to not inherit from class `Slot`.

For brevity, we may use the uncapitalized names interchangeably when referring to instances of these classes (e.g., “product slot”, “qualifier”, etc.) in the rest of this thesis.

4.4 Framework Layer

The framework layer defines and enforces the standard processes and functionality of the framework. It controls the current analysis stage, defines how to communicate with differ-

ent framework components, and manages slots and constraints in a centralized place. The behaviour of this layer should be general enough to orchestrate different type systems, and type system developers can focus on their development without delving into the layer's technical details.

4.4.1 Analysis Order

The order in which dataflow analysis is performed on different CFGs can affect a type system's overall accuracy and precision. This is because the analysis results on one CFG may help the analysis of another CFG. One example is that the type of a final field can be refined by checking its initializer, which then benefits the analyses on internal methods that use this field. Another example is analyses for both lambda expressions and local classes require the types of the variables they captured. Lambda expressions and local classes declared inside a method may capture the local variables and parameters of the method, but each of these constructs is analyzed as a separate CFG. Generally speaking, we need the types of certain variables to improve the dataflow analyses of the CFGs that use these variables. A limitation of this strategy is the more precise information is only available to members of the current compilation unit, as other compilation units may not have been fully initialized by the compiler.

This idea of reordering the analysis targets is originated and implemented in the Checker Framework[14]. In particular, the implementation traverses the compilation unit tree and performs the analysis in level order. Traversing in level order has the disadvantage of losing the enclosing relations between different structures in code. This means at each level, it needs to accumulate the analysis results and further propagate everything to the next level, even if some results have become redundant as it finishes analyzing an inner scope. The implementation causes some extra memory footprint and may leak analysis results to other CFGs.

In UniFlow, we propose a preorder traversal to address the issues. As shown in Algorithm 1, we can implement our solution in the common visitor pattern. When it finds a class tree, the SortClassMembers function sorts the class members and visits them in the following order:

1. static fields and static initializers

[14] https://github.com/eisop/checker-framework/blob/e6c069537bd8dae1c79c70665859c36f382c8d0c/framework/src/main/java/org/checkerframework/framework/type/GenericAnnotatedTypeFactory.java#L1300

Algorithm 1 Perform Analysis in Defined Order

```
1:  function VISIT(Tree tree)
2:      childTrees ← tree.getChildren()
3:      if tree is a class declaration then
4:          childTrees ← SORTCLASSMEMBERS(childTrees)
5:      else if tree is a field initializer, initializer block, method declaration, or lambda
    expression then
6:          PROCESSCFG(BUILDCFG(tree))
7:      end if
8:
9:      for each childTree in childTrees do
10:         VISIT(childTree)
11:     end for
12:     RESULTSCLEANUP(tree)
13: end function

14: function PROCESSCFG(ControlFlowGraph cfg)
15:     PERFORMDATAFLOWANALYSIS(cfg)
16:     PERFORMTYPECHECKS(cfg)
17: end function
```

2. instance fields and instance initializers
3. constructors
4. other methods

This ensures the analysis on field initialization always happens before its usage in the current class. When the visitor finds a field initializer, an initializer block, a method declaration, or a lambda expression, it will generate a corresponding CFG and start processing the CFG. As defined in the ProcessCFG function at line 14, the function performs dataflow analysis and immediately uses the resolved types to execute type checks. After analyzing the CFG of the current tree, the visitor proceeds to visit the children of this tree. If the visitor finds some other tree, it will traverse its children to continue finding the previously mentioned trees. For instance, there can be local class declarations inside a method declaration.

When the algorithm finally reaches line 12, analysis results from the current scope may have become redundant, so it attempts to clean up the analysis results. Some fine-tuning is required for the ResultsCleanup function. For example, analysis results from a field should only be removed after visiting its enclosing class.

4.4.2 Slot Location

Section 4.1 explains the importance of tracking the location of each created slot. In Checker Framework Inference, this is achieved by requiring a location object to create a new slot. However, the location of a declaration construct cannot be determined until its enclosing compilation unit is processed. For example, if javac processes files in the order of "A.java" followed by "B.java" and class A accesses a field in class B, compiler plugins will not have the AST to determine the location of B's field when analyzing components in A. This can introduce mutable or missing locations to the framework. Moreover, the `VariableAnnotator`[15] in Checker Framework Inference performs location search and type resolution simultaneously, further exacerbating the readability and complexity.

In UniFlow, the creation of each slot is different according to the type of location it requires. Fortunately, only source slots require accurate locations because their solutions can be inserted back. Other slots are only created for local expressions in a dataflow

[15] https://github.com/eisop/checker-framework-inference/blob/47bf3c36b99bfeb4d86ab69857c40033ea774c7c/src/checkers/inference/VariableAnnotator.java

analysis, and their locations are used for debugging. Therefore, the new strategy for locating slots can be clarified as follows. The type resolver is responsible for determining the appropriate slot to use and requesting the slot from the slot manager. If the slot is a source slot, the slot manager will not ask the type resolver to provide a location. Instead, it will traverse the compilation unit tree in a separate stage to match the source slots accordingly. If the slot is not a source slot, the slot manager will require the current node to deduce the best approximation of the location.

4.4.3 Multiple Type Systems Support

The framework layer needs to ensure each component supports multiple type systems. Furthermore, the framework allows each type system to own multiple qualifier hierarchies, although most type systems in the Checker Framework only have one qualifier hierarchy. This section explores our design to provide the necessary foundation.

Type Resolution Stage

Type resolution consists of resolving the type of declaration constructs and resolving the type of CFG nodes. When handling the CFG nodes, resolved types accumulator must collect analysis results from different type systems, and each type system can focus on the qualifiers they support. Algorithm 2 shows how this stage is generally managed.

The resolved types accumulator itself can be considered as a transfer function, with the obligation to delegate the call to each type system and combine their results. The function VisitNode shows how it handles each node in the CFG. Here, we provide descriptions for the important lines in the function.

Line 2 We declare a variable r to store the transfer results we have accumulated. Its initial value is null.

Line 3 We iterate through type systems in a topological order based on their dependencies on each other. This is because the result from a type system may improve the result of another type system, such as the Initialization Checker in Nullness Checker.

Line 4 We call the transfer function of the current type system, with transfer input and accumulated results r provided. The result of this function is assigned to r'. The result uses qualified types to represent annotated types and product slots to store their qualifiers. This ensures a unified communication method, while each type system can have multiple qualifier hierarchies.

Algorithm 2 Resolved Types Accumulator

```
 1: function VisitNode(node, input)
 2:     r ← null                                          ▷ Declare a transfer result variable
 3:     for each t in type systems sorted in topological order do
 4:         r′ ← callTransfer(t, node, input, r)
 5:         r ← combine(r, r′)                  ▷ Combine existing results with the new result
 6:     end for
 7:     return r
 8: end function

 9: function Combine(r, r′)
10:     if r is null then
11:         return r'
12:     end if
13:     newValue ← combination of the result values in r and r′
14:     newThenStore ← combination of the "then" stores in r and r′
15:     newElseStore ← combination of the "else" stores in r and r′
16:     return a new transfer result from newValue, newThenStore and newElseStore
17: end function
```

Line 5 We combine r and r' and assign the combined result back to r.

The function Combine will combine results from different type systems by merging their respective components. Let K be the set of possible Java expressions, V be the set of possible qualified types using product slots, and $\Sigma : K \to V$ be a store that maps a Java expression to its refined qualified type. Assume each $v \in V$ is represented as $p\ T$, where $p = (s_1, \ldots, s_n)$ is a product slot, and T is the Java type identifier. We can consider a transfer result as $r = \{rv, \Sigma_1, \Sigma_2\}$, where $rv \in V$ is the result value of the current node, Σ_1 and Σ_2 are then and "else" stores respectively. Note that it is possible that the result only has one store, which can be covered by the special case $\Sigma_1 = \Sigma_2$. Finally, we can formalize the combining process as follows:

$$combine(r, r') = \{combine(rv, rv'), combine(\Sigma_1, \Sigma_1'), combine(\Sigma_2, \Sigma_2')\} \tag{4.1}$$

$$combine(\Sigma, \Sigma') = \{\alpha \mapsto combine(\Sigma(\alpha), \Sigma'(\alpha)) \mid \alpha \in K\} \tag{4.2}$$

$$combine(p\ T, p'\ T) = combine(p, p')\ T \tag{4.3}$$

$$\begin{aligned} combine(p, p') &= combine((s_1, \ldots, s_n), (s_1', \ldots, s_m')) \\ &= (s_1, \ldots, s_n, s_1', \ldots, s_m') \end{aligned} \tag{4.4}$$

The reason for having identical Ts in equation 4.3 is that the framework does not refine Java types, which is consistent with Java's type system and simplifies the combining process. We also assumed $v \in V$ is represented as $p\ T$ for the brevity of the formulas. In reality, a qualified type can be a nested structure (e.g., array type or generic type), and we need to combine the matching product slots recursively.

The strategy to handle declaration constructs is very similar. We can replace the node visitor with a tree visitor focusing on those declarations. Instead of calling the transfer function, we invoke the type resolver of each type system to get an instance of a qualified element. After that, we can combine the product slots of multiple qualified elements.

Constraint Generation Stage

There is no predefined order to execute type systems in the constraint generation stage, as we already have the types resolved for all the nodes and declarations. Each type system may access all the qualifiers in product slots and generate its constraints independently. Since all the constraints are based on slots, the framework provides a utility class to return a proper constraint for each requested type relation. The utility class will try to eagerly

evaluate the type relation when possible. For example, if we need a subtype constraint between two constant slots, the utility class will ask the corresponding type system for the answer to produce a solved constraint (i.e., a constant instance indicating a true or false result).

The constraint manager is responsible for storing all the generated constraints. It can output the constraints of a type system in the subsequent stages. It also provides error messages to the analysis output, which will be explained in the next section.

Constraints Solving Stage

Algorithm 3 demonstrates an overview of the contraint solving procedure. The algorithm starts with declaring an empty mapping s to store solutions. It then begins to iterate through type systems by the topological order of their dependency graph. The following actions are performed in each iteration for the current type system t:

Line 3 We get the constraints generated by t from constraint manager and assign the set to c.

Line 4 We substitute any solved variables in c with its solution in s. Note that each type system may be in either type checking or inference mode. If an ancestor is in type checking mode, we already have its solutions in the type resolution stage. In other words, only inference solutions are effective to s and this substitution.

Line 5 This step will initiate the process of encoding constraints, executing an external solver, and decoding the solutions. The details are omitted because we can safely reuse the existing strategy from Checker Framework Inference. If the constraints cannot be solved because they are unsatisfiable or an ancestor causes cascading failures[16], we can assume $s' = \emptyset$.

Line 6 We update our solutions with the union of s and s'.

Compared to solving all constraints at once, the benefits of this algorithm are twofold. First, any inference failures in one type system will not affect other type systems that are not effectively depending on it, making the analysis results more accurate and debuggable.

[16]Such failures can happen if there exist real constraint variables owned by other type systems in c, whose solutions cannot be determined in any previous steps.

Algorithm 3 Solve Constraints in Defined Order

```
1: s ← {}                                   ▷ Declare a mapping of solved variable to its solution
2: for each t in type systems sorted in topological order do
3:     c ← getConstraints(t)
4:     c ← substituteSolvedVariables(c, s)
5:     s′ ← solveConstraints(t, c)
6:     s ← s ∪ s′
7: end for
```

Second, each type system has the flexibility to choose its solver and encoding/decoding strategies, making the logic simpler and more modular.

A disadvantage of this algorithm is the inference results from one type system may affect the analyses of its descendants, for the results can be either too loose or too strict for the descendants. The possible solutions include introducing soft constraints or manually tuning the inference results. We leave this as a topic for future study.

4.4.4 Analysis Messages

As outlined in our design goals, we want UniFlow to underpin a friendly message reporting mechanism. Since the solution to many constraints cannot be determined when they are generated, we decided to make the message a detached property of constraint that can be queried later. In other words, for a specific type rule application at a program location, the type system may generate a set of constraints and a message to show when the constraints are unsatisfiable, which will be managed by the constraint manager. Once the solver has finished its work, the framework can retrieve and output the corresponding failure messages to users. The reason for it being a detached (i.e., optional) property is that there are some naturally derived constraints without explanations. For example, we may generate merge slots when merging stores from different edges, where each LUB/GLB is always the supertype/subtype of the types they merged.

In Section 4.3, we defined the three components of a message: message kind, code position and message body. The type system should specify message kind and body, and the framework can retrieve code position from the current CFG node or AST tree. In addition to message generation, we considered the potential issues in message reporting. According to Dataflow Framework's manual [1], the same expression may appear in multiple blocks, which is caused by special control flows such as try-finally statements. This means we may observe duplicate messages for the same issue at the same location, so the framework must

deduplicate messages before producing outputs. Another potential issue is that storing all the messages in memory is not scalable, for the number of messages positively correlates to the source code's size and the number of constraints. One simple solution is to use constant string templates consistently, so we only need to store formatting arguments and compute the actual messages later. Eventually, a more sophisticated solution is to compute and offload all messages to a local disk-based database, such as H2 MVStore[17] or MapDB[18].

4.5 Type System Layer

The type system layer provides the basic architecture and functionality for type system developers. It defines the essential components and the general implementation of a type system, which developers can override to achieve different purposes. The goal of this layer is to properly define the obligations of a type system, while giving it the flexibility to extend and customize its behaviour.

There are seven basic components in a type system. We have listed five of them in Section 4.1. The rest of the components are qualifier hierarchy and type hierarchy. A type system may define one or multiple qualifier hierarchies. A qualifier hierarchy is the implementation of a pluggable type lattice. Compared to the `QualifierHierarchy`[19] in the Checker Framework, it only represents a single hierarchy with one top qualifier and one bottom qualifier. It offers various utilities for the covered qualifiers, such as determining subtype relations, finding the LUB or GLB of two qualifiers, and converting an `AnnotationMirror` to the corresponding qualifier. Note that the qualifier hierarchy operates on real qualifiers and does not recognize constraint variables. On the other hand, type hierarchy aims to generate constraints for the relations between qualified types, which is similar to the `TypeHierarchy`[20] in the Checker Framework. In most cases, it should obey the rules defined in the Java Language Specification [13].

In the following sections, we will explore how a type system works in type resolution and constraint generation, with some formalizations for illustrative purposes only. We omit the explanations for constraint encoding and solution decoding because the related strategies have been studied on Checker Framework Inference [12, 20].

[17] https://www.h2database.com/html/mvstore.html

[18] https://mapdb.org/

[19] https://github.com/eisop/checker-framework/blob/1b92dc5b55f25c061c963cbb14f0876748972d1d/framework/src/main/java/org/checkerframework/framework/type/QualifierHierarchy.java

[20] https://github.com/eisop/checker-framework/blob/1b92dc5b55f25c061c963cbb14f0876748972d1d/framework/src/main/java/org/checkerframework/framework/type/TypeHierarchy.java

$$
\begin{array}{rlrl}
P & ::= \overline{Cls} & & \\
Cls & ::= \texttt{class}\ Cid\ \texttt{extends}\ T\ \{\ \overline{fd}\ \overline{md}\ \} & fd & ::= T\ f; \\
md & ::= T_r\ m(\overline{T\ pid})\ \{\ stmt\ \} & T & ::= q\ C \\
stmt & ::= \texttt{return}\ e & q & ::= qid \mid \alpha \mid \epsilon \\
e & ::= \texttt{null} \mid x \mid \texttt{new}\ T() \mid e.f \mid e_0.f := e_1 \mid & C & ::= Cid \mid \texttt{Object} \\
 & \quad e_0.m(\overline{e}) \mid (T)\ e & x & ::= pid \mid \texttt{this} \\
pid & \text{parameter identifier} & f & \text{field identifier} \\
m & \text{method identifier} & Cid & \text{class identifier} \\
qid & \text{concrete qualifier identifier} & \alpha & \text{constraint variable identifier}
\end{array}
$$

Figure 4.4: Syntax of our programming language

4.5.1 A Java-Like Programming Language

In Figure 4.4, we define a simple Java-like programming language that we can use for some formalizations in later sections. This language is a modified version of the one that is used in the first paper of Checker Framework Inference [12].

A program P is formed by one or multiple class declarations. Each class declaration starts with its name and superclass, followed by a block of field and method declarations. A field declaration consists of a type and its name. A method declaration has a return type, a method identifier, some parameters, and a statement in the method body. The statement returns the value of an expression e. The possible forms of the expression e include null literal, parameter read, object creation, read/write of a field, method invocation, and explicit type cast.

A type is a qualifier followed by a class name. In the source code, the qualifier represents an optional annotation on the type, which can only be a concrete qualifier from the lattice. For the type system, the qualifier must be a product slot containing contraint variables, where some variables are solved and some are yet to be solved.

Since this language is a subset of Java, we may use the definitions in Dataflow Framework's manual to convert method bodies into CFGs [1]. Specifically, we list the CFG nodes that are involved[21] in Figure 4.5. Note that x can indicate the method receiver `this`, and

[21] Actually, we ignored `ClassNameNode` for type expression T because there is little context to evaluate the node itself.

Node Expression	**Node Type**
`null`	NullLiteralNode
$e.f$	FieldAccessNode
$e.m$	MethodAccessNode
x	LocalVariableNode
$e_0.m(\overline{e})$	MethodInvocationNode
$e_0.f := e_1$	AssignmentNode
`new` $T()$	ObjectCreationNode
$(T)\ e$	TypeCastNode
`return` e	ReturnNode

Figure 4.5: Relevant CFG nodes to our programming language

we treat it as a final local variable access. In reality, there is a specific node to represent the receiver.

Both expression and subexpression can have a corresponding node. In other words, translating an expression in AST can result in multiple nodes, starting from the first subexpression, in their evaluation order. For example, the nodes generated for the expression `this.x.y` are `this`, `this.x` and `this.x.y`, where the last node is the corresponding node.

4.5.2 Type Resolution Stage

The type resolver is responsible for handling both declaration constructs and CFG nodes. To improve modularity and maintainability, we decompose it into two components: a declaration type resolver for the types in declarations and a node type resolver for the types of CFG nodes.

Declaration Types Resolution

The declaration type resolver is an element visitor that accepts an element, an abstraction of a declared entity generated by javac. It will introduce constraint variables to the types in the element and returns a qualified element. In type checking mode, all constraint variables are constants determined by the existing annotations or defaulting strategy. In type inference mode, constant slots are only used for existing annotations or bytecode locations, while source slots are created for other places.

We can formalize the declaration type resolver based on our language to demonstrate the framework's default implementation.

$$
\text{declType}(E) = \begin{cases} \texttt{class}\ Cid\ \texttt{extends}\ \text{qType}(E, T) & \text{if } E \text{ is a class declaration} \\ \text{qType}(E, T_r)\ m(\overline{\text{qType}(E, T)\ pid}) & \text{if } E \text{ is a method declaration} \\ \text{qType}(E, T)\ f & \text{if } E \text{ is a field declaration} \end{cases}
$$

$$
\begin{aligned}
\text{qType}(E, T) &= \text{qType}(E, q\ C) \\
&= \begin{cases} \alpha_{\text{const}}(q)\ C & \text{if } q \neq \epsilon \text{ and is in the lattice} \\ \alpha_{\text{const}}(\text{defaultQual}(C))\ C & \text{if type checking mode or } E \text{ is not from source} \\ \alpha_{\text{src}}\ C & \text{otherwise} \end{cases}
\end{aligned}
$$

In the above formalization, we defined $\text{declType}(E)$ as the function that consumes an element E and returns the corresponding qualified element. It searches for type uses in the element and replaces each T with $\text{qType}(E, T)$. The $\text{qType}(E, T)$ is a helper function to ensure we have a qualified type. If T is already annotated with a valid annotation, qType will use a constant slot to wrap the corresponding concrete qualifier. The function then checks if the type system is in type checking mode or E is not from the source code. If either is true, we ask for the default concrete qualifier for class C and convert it into a constant slot; otherwise, we introduce a source slot as the constraint variable for inference. The behaviour of the function defaultQual(C) should be specified by developers. It can be as simple as returning the top qualifier, or it can consider different factors, such as the bounds of C and the type use context.

Note that the function declType always returns the same qualified element for the same input. It memorizes the previous results so that it will not create new slots for the resolved elements.

We assumed that the type system has only one qualifier hierarchy. This assumption simplifies our explanation by safely ignoring trivial actions such as constructing product slots or adding loops. For brevity, we will continue to make this assumption throughout the rest of Section 4.5.

Node Types Resolution (Transfer Function)

Resolving the type of a CFG node requires the coordination of the transfer function and the node type resolver. The transfer function manages stores and constructs transfer results,

while the node type resolver is responsible for deducing the qualified type of the current node. Both are implementations of a node visitor to handle different kinds of nodes.

We can formalize the basic transfer rules as shown in Figure 4.6. Recall that the domain of Γ is the set of nodes, whereas the domain of Σ is the set of Java expressions. The key difference between a node and a Java expression is that the node is only a single instance of expression. For example, for the expression $\texttt{this}.m(\texttt{this}.f, \texttt{this}.f)$ in AST, there will be two distinct nodes for the two $\texttt{this}.f$ arguments, but both arguments are the same in Java expression.

We also employ a few helper functions. We use shouldStore(e) to determine if the expression corresponds to node e should be accepted by stores. Typically, it holds for local variable accesses, field accesses, and invocations of pure methods. The function merge(Σ_1, Σ_2) computes the LUB of the two stores. The removeRelated(Σ, e) is similar to `removeConflicting` in the Checker Framework[22], which is a technique for handling gen/kill problems. It notifies the store Σ that the type of e will be updated, implying any results related to the type of e should be removed. For example, if the type of x needs to be updated, then the existing type of $x.y$ is no longer valid.

We now explain each rule in Figure 4.6. For an assignment node $e_0.f := e_1$, we resolve the qualified type of the node as T, then we ensure the node has type T in Γ and refine the type of the LHS expression $e_0.f$ to T in Σ''. The outputting "then" and "else" stores are both Σ'', which is formed by merging the input stores, removing no longer valid mappings, and adding the new mapping. For any expression e that is not an assignment node, we need to consider if shouldStore(e) holds and if the resolved type is different from the one in Σ. If both conditions are true, we apply the rule DefaultStore, which is similar to Assign but refines the type of the entire expression e in Σ''; otherwise, we apply the last rule Default to store the qualified type of e in Γ and propogate the merged store Σ. Despite of the formatting, these rules are different from any formal type rules, for the transfer function may need to be applied several times in fixed-point iterations.

We reiterate that the formalizations are to illustrate the basic logic, and developers can extend each component for their approaches. For instance, the stated rules unsoundly assume all method calls are pure. To resolve this issue, we need to introduce a basic type system, "PureChecker", that determines if a method is side-effect-free and if it is deterministic. Then another type system based on the "PureChecker" (in type checking mode) may override the logic to clear the stores after a non-side-effect-free method call.

[22] https://github.com/eisop/checker-framework/blob/1b92dc5b55f25c061c963cbb14f0876748972d1d/framework/src/main/java/org/checkerframework/framework/flow/CFAbstractStore.java#L910

$$\textsc{Default}\frac{\begin{array}{c} e \text{ is not an assignment} \\ \Sigma = \text{merge}(\Sigma_1, \Sigma_2) \\ \Gamma, \Sigma \vdash [\![e]\!] = T \\ \text{not shouldStore}(e) \text{ or } \Sigma(e) = T \end{array}}{\Gamma, \Sigma_1, \Sigma_2 \vdash e : \Gamma\left[e \mapsto T\right], \Sigma, \Sigma}$$

$$\textsc{DefaultStore}\frac{\begin{array}{c} e \text{ is not an assignment} \\ \Sigma = \text{merge}(\Sigma_1, \Sigma_2) \\ \Gamma, \Sigma \vdash [\![e]\!] = T \\ \text{shouldStore}(e) \qquad \Sigma(e) \neq T \\ \Sigma' = \text{removeRelated}(\Sigma, e) \\ \Sigma'' = \Sigma'\left[e \mapsto T\right] \end{array}}{\Gamma, \Sigma_1, \Sigma_2 \vdash e : \Gamma\left[e \mapsto T\right], \Sigma'', \Sigma''}$$

$$\textsc{Assign}\frac{\begin{array}{c} \Sigma = \text{merge}(\Sigma_1, \Sigma_2) \\ \Gamma, \Sigma \vdash [\![e_0.f := e_1]\!] = T \\ \Sigma' = \text{removeRelated}(\Sigma, e_0.f) \\ \Sigma'' = \Sigma'\left[e_0.f \mapsto T\right] \end{array}}{\Gamma, \Sigma_1, \Sigma_2 \vdash e_0.f := e_1 : \Gamma\left[e_0.f := e_1 \mapsto T\right], \Sigma'', \Sigma''}$$

Figure 4.6: Abstract transfer rules for the considered CFG nodes. Γ is the type environment that maps each node e to its corresponding qualified type, and Σ_1/Σ_2 are the "then"/"else" stores respectively. In the premises, we use $\Gamma, \Sigma \vdash [\![e]\!] = T$ to represent the qualified type of e is T from the node type resolver. The conclusions are of the form $\Gamma, \Sigma_1, \Sigma_2 \vdash e : \Gamma', \Sigma'_1, \Sigma'_2$, which means, given a node e and the satisfied premises, the transfer function consumes the inputs $\Gamma, \Sigma_1, \Sigma_2$ and returns the updated environments $\Gamma', \Sigma'_1, \Sigma'_2$.

"PureChecker" will also complement the logic of the function shouldStore(e) to reject non-pure method invocations.

Another example is the Nullness Checker in the Checker Framework, which overrides the transfer function to handle a null check node. The logic can be formalized as follows:

$$\textsc{NullCheck}\frac{\begin{array}{c}\Sigma = \text{merge}(\Sigma_1, \Sigma_2) \\ \Gamma, \Sigma \vdash [\![e \neq \texttt{null}]\!] = T \\ \Sigma_1' = \text{removeRelated}(\Sigma_1, e) \\ \Sigma_2' = \text{removeRelated}(\Sigma_2, e) \\ \Sigma_1'' = \Sigma_1' \left[e \mapsto \text{makeNonNull}(\Sigma_1(e))\right] \\ \Sigma_2'' = \Sigma_2' \left[e \mapsto \text{makeNullable}(\Sigma_2(e))\right]\end{array}}{\Gamma, \Sigma_1, \Sigma_2 \vdash e \neq \texttt{null} : \Gamma\left[e \neq \texttt{null} \mapsto T\right], \Sigma_1'', \Sigma_2''}$$

This rule has two objectives. First, it needs to set the qualified type of the node in Γ. Second, it refines the type of the expression e in both stores. "Then" store Σ_1'' maps to the type when $e \neq$ `null` evaluates to true, which is achieved by changing the primary qualifier of $\Sigma_1(e)$ to α_{const}(`@NonNull`). "Else" store Σ_2'' maps to the type when $e \neq$ `null` evaluates to false, so we change the primary qualifier of $\Sigma_2(e)$ to α_{const}(`@Nullable`).

Node Types Resolution (Node Type Resolver)

Next, we will formalize the node type resolver, i.e., the rules for $[\![e]\!]$. Before analyzing a CFG, the resolver can collect the known facts to prepare an input store for the entry block. For example, analyzing a lambda function requires the types of all captured variables. For this simple language, the resolver can prepare the types of parameters and receiver `this` before entering a method's CFG. The types of parameters can simply be extracted from the qualified element of the method. The type of the receiver `this` depends on the strategy of a specific type system.

Figure 4.7 gives the basic rules for the node type resolver. Our rules assume the input stores for the entry block already contain types of local variable expressions. Before discussing the details, we explain the new helper functions introduced. The element function consumes a field access $e.f$ or method access $e.m$ and retrieves the associated field or method declaration as a javac element. Viewpoint adaptation is an abstract function that adapts the type of a field or method access according to the type of its receiver [11]. We use the notation $\triangleright$ for this function, where the left operand is the receiver's type, and the right operand is the type to be adapted. assignQual(α_{RHS}) decides the qualifier that applies to the type of an assignment node. In type checking mode, it returns α_{RHS}; in type inference mode, it returns the refinement slot α_{refine} corresponds to this assignment node.

$$\text{T_FieldAcc}\ \frac{\Sigma(e.f) = T}{\Gamma, \Sigma \vdash [\![e.f]\!] = T} \qquad \text{T_FieldAccNew}\ \frac{\Sigma(e.f) = \emptyset \qquad \Gamma(e) = T_e \quad \text{declType}(\text{element}(e.f)) = T_d\ f \quad T_e \rhd T_d = T}{\Gamma, \Sigma \vdash [\![e.f]\!] = T}$$

$$\text{T_Return}\ \frac{\Gamma(e) = T}{\Gamma, \Sigma \vdash [\![\texttt{return}\ e]\!] = T} \qquad \text{T_Null}\ \frac{T = \text{defaultQual}(\texttt{nulltype})\ \texttt{nulltype}}{\Gamma, \Sigma \vdash [\![\texttt{null}]\!] = T}$$

$$\text{T_MthAcc}\ \frac{\Gamma(e) = T_e \quad \text{declType}(\text{element}(e.m)) = T_{rd}\ m(\overline{T_d\ pid}) \quad (T_{rd}\ m(\overline{T_d\ pid}))[\alpha_{\text{poly}}/\texttt{poly}] = T'_{rd}\ m(\overline{T'_d\ pid}) \quad T_e \rhd T'_{rd}\ m(\overline{T'_d\ pid}) = T_r\ m(\overline{T\ pid})}{\Gamma, \Sigma \vdash [\![e.m]\!] = T_r\ m(\overline{T\ pid})}$$

$$\text{T_MthInvk}\ \frac{\Gamma(e_0.m) = T_r\ m(\overline{T\ pid})}{\Gamma, \Sigma \vdash [\![e_0.m(\overline{e})]\!] = T_r} \qquad \text{T_LocalVarAcc}\ \frac{}{\Gamma, \Sigma \vdash [\![x]\!] = \Sigma(x)}$$

$$\text{T_NewObj}\ \frac{}{\Gamma, \Sigma \vdash [\![\texttt{new}\ T()]\!] = \text{qType}(T)} \qquad \text{T_Cast}\ \frac{}{\Gamma, \Sigma \vdash [\![(T)\ e]\!] = \text{qType}(T)}$$

$$\text{T_Assign}\ \frac{\text{declType}(\text{element}(e_0.f)) = T_{LHS}\ f \qquad T_{LHS} = \alpha_{LHS}\ C_{LHS} \quad \Gamma(e_1) = T_{RHS} \qquad T_{RHS} = \alpha_{RHS}\ C_{RHS} \quad T = \text{assignQual}(\alpha_{RHS})\ C_{LHS}}{\Gamma, \Sigma \vdash [\![e_0.f := e_1]\!] = T}$$

Figure 4.7: Abstract type resolution for the considered CFG nodes. Each rule for node e can be considered as a case in the `visit`(e) method. Γ is the input type environment from the transfer function. $\Sigma = \text{merge}(\Sigma_1, \Sigma_2)$ is the merged input stores from the transfer function. For any subnode e' in the node e, we denote a look-up of the type of the subnode as $\Gamma(e') = \ldots$. Since e' is always evaluated before e in a CFG, the look-up does not require extra arguments and is guaranteed to succeed.

We begin our explanation with the difference between T_FIELDACC and T_FIELDACCNEW. For a field access $e.f$ whose type already exists in Σ, we apply T_FIELDACC to directly reuse the existing qualified type. If we cannot find an existing type for $e.f$ in Σ, then T_FIELDACCNEW is applied. To get the qualified type of $e.f$, we need to get the type of the receiver ($[\![e]\!]$) and the declared type of the field (T_d), and we perform a viewpoint adaptation $T_e \rhd T_d$.

It is also necessary to explain the rules T_MTHACC, T_LOCALVARACC and T_ASSIGN, while other rules are rather straightforward. T_MTHACC applies to a method access node. We first look up the declared type of the method $T_{rd}\ m(\overline{T_d\ pid})$. Then we substitute all polymorphic qualifiers in the declared type with a polymorphic instance slot α_{poly} dedicated to this location. This ensures we create a fresh α_{poly} for each call site of a polymorphic method. Lastly, we obtain the final resolved type by performing a viewpoint adaptation.

T_LOCALVARACC loads the type of a local variable $[\![x]\!]$ directly from Σ. The reason is that we have prepared the types of local variables, and this simple language does not support reassigning the value of a local variable. In reality, we must consider declarations and reassignments of local variables in a CFG, and the strategy is akin to how we handle field declarations and reassignments.

T_ASSIGN handles the type of an assignment node, or more specifically, the refined type of the LHS after assigning the RHS. This rule requires the declared type of the LHS $e_0.f$ and the qualified type of the RHS e_1. We extract two important pieces of information from these types: C_{LHS} and α_{RHS}. As previously discussed, we do not refine any Java types to simplify the process of combining and merging different stores, so C_{LHS} will be the class identifier in the final resolved type T. The qualifier of T is obtained from the helper function $\text{assignQual}(\alpha_{RHS})$. Although it is possible that T_{RHS} is not a subtype of T_{LHS}, we leave the validation part to the type checker.

4.5.3 Constraint Generation Stage

Similar to how we split the work of type resolution, the type checker is also split into a declaration type checker and a node type checker, where the former handles constraints on declaration constructs and the latter handles constraints on CFG nodes.

Constraints for Declarations

The declaration type checker is an element visitor that focuses on class, field, and method declarations. Let $K(X)$ be the set of constraints generated for declaration X. Figure 4.8

$$\text{C_Class}\ \frac{\begin{array}{c}\text{declType}(\texttt{class}\ Cid\ \texttt{extends}\ T) = \texttt{class}\ Cid\ \texttt{extends}\ T' \\ K_1 = \text{wellformed}(T') \\ T' = \alpha'\ C' \qquad K_2 = \text{upperbound}(Cid) <: \alpha'\end{array}}{\vdash K(\texttt{class}\ Cid\ \texttt{extends}\ T) = K_1 \cup K_2}$$

$$\text{C_Field}\ \frac{\begin{array}{c}\text{declType}(T\ f) = T'\ f \\ K_1 = \text{wellformed}(T')\end{array}}{\vdash K(T\ f) = K_1}$$

$$\text{C_Method}\ \frac{\begin{array}{c}\text{declType}(T_r\ m(\overline{T\ pid})) = T'_r\ m(\overline{T'\ pid}) \\ K_1 = \text{wellformed}(T'_r) \qquad K_2 = \text{wellformed}(\overline{T'}) \\ K_3 = \text{override}(T'_r\ m(\overline{T'\ pid}))\end{array}}{\vdash K(T_r\ m(\overline{T\ pid})) = K_1 \cup K_2 \cup K_3}$$

Figure 4.8: Abstract constraint generation for the considered declarations

illustrates our fundamental rules.

Again, we have introduced a few helper functions. The function upperbound(C) returns the qualifier upper bound for the Java type C, whose actual determination strategy is type system dependent. The function wellformed(T) generates a set of constraints for the well-formedness check of qualified type T. Its basic logic can be formalized as follows:

$$\text{wellformed}(T) = \text{wellformed}(\alpha\ C) = \{\alpha <: \text{upperbound}(C)\}$$

The function override($T'_r\ m(\overline{T'\ pid})$) generates constraints for method overriding. Let $T'_r\ m(\overline{T'\ pid}) = T'_r\ m(T'_1\ p_1, \ldots, T'_n\ p_n)$. If m overrides m_s in the superclass, let the qualified element of m_s be

$$\text{declType}(T_{sr}\ m_s(\overline{T_s\ pid})) = T'_{sr}\ m_s(\overline{T'_s\ pid}) = T'_{sr}\ m_s(T'_{s1}\ p_{s1}, \ldots, T'_{sn}\ p_{sn})$$

We will have the the following constraints:

$$\text{override}(T'_r\ m(\overline{T'\ pid})) = \{T'_{si} <: T'_i \mid i \in \mathbb{Z}, 1 \leq i \leq n\} \cup \{T'_r <: T'_{sr}\}$$

If m does not override any methods, the function returns an empty set.

Given the explanations for the helper functions, the three rules in the figure are straightforward. It is worth noting that K_2 in C_Class is the inheritance rule for class declaration, which ensures the qualifier upper bound of the class Cid is a subtype of the qualifier α' in its qualified superclass T'.

Constraints for Nodes

The node type checker is a node visitor that applies type rules to each CFG node. Let $K(e)$ be the set of constraints generated for node e. Figure 4.9 illustrates our fundamental rules. We have one new helper function, currentMethod(), which simply returns the method associated with the current CFG.

There are no explicit constraints for null literal, field access, method access, or local variable access. So we will focus on explaining other rules.

C_NewObj checks the type of a new object instance is well-formed. C_Cast ensures the casted type is well-formed and is comparable to the original type. The notation $<:>$ represents the comparable constraint introduced in the GUT paper [12], which means one operand should be the subtype of the other. C_Assign verifies that, for an assignment node, the type of the RHS is a subtype of the declared type of the LHS. C_Return checks the type of the returning node is a subtype of the method's declared return type. Finally, C_MthInvk produces the constraints for a method invocation. It ensures the type of each argument is the subtype of the matching parameter's type.

Recall that the constraint variables are slots, but we have listed some constraints using qualified types in the form of T_1 *op* T_2. For these rules, type hierarchy is the component to help with generating concrete constraints. As previously stated, it follows the subtyping strategy defined for Java types in the language specification [13], and it applies the strategy to qualifiers in T_1 and T_2. There are no constraints for Java types because it is the compiler's responsibility to ensure their type safety.

Implicit Constraints from Slots

Despite the constraints from declarations and nodes, there are some implicit constraints derived from slots. Specifically, for the slots we have introduced, we must have constraints on the refinement and merge slots. The implicit constraints may not require error messages since they cannot be treated as the root cause of type checking or inference failures.

Given a refinement slot α_{refine} that indicates the refined type of an assignment. If the RHS's type has qualifier α_{RHS}, an equality constraint

$$K(\alpha_{\text{refine}}) = \{\alpha_{\text{refine}} = \alpha_{RHS}\}$$

needs to be established.

$$\text{C_NewObj}\ \frac{\Gamma(\texttt{new}\ T()) = T' \quad K_1 = \text{wellformed}(T')}{\Gamma \vdash K(\texttt{new}\ T()) = K_1} \qquad \text{C_Cast}\ \frac{\Gamma(e) = T' \quad \Gamma((T)\ e) = T'' \quad K_1 = \text{wellformed}(T'') \quad K_2 = T' <:> T''}{\Gamma \vdash K((T)\ e) = K_1 \cup K_2}$$

$$\text{C_Assign}\ \frac{\text{declType}(\text{element}(e_0.f)) = T_{LHS}\ f \quad \Gamma(e_1) = T_{RHS} \quad K_1 = T_{RHS} <: T_{LHS}}{\Gamma \vdash K(e_0.f := e_1) = K_1}$$

$$\text{C_Return}\ \frac{\Gamma(\texttt{return}\ e) = T' \quad \text{declType}(\text{currentMethod}()) = T_r\ m(\overline{T\ pid}) \quad K_1 = T' <: T_r}{\Gamma \vdash K(\texttt{return}\ e) = K_1}$$

$$\text{C_MthInvk}\ \frac{\Gamma(e_0.m) = T_r\ m(\overline{T\ pid}) \quad \overline{T} = T_1, \ldots, T_n \quad \overline{e} = e_1, \ldots, e_n \quad \Gamma(e_i) = T_i' \quad K_i = T_i' <: T_i}{\Gamma \vdash K(e_0.m(\overline{e})) = \bigcup_{i=1}^{n} K_i}$$

$$\text{C_Null}\ \frac{}{\Gamma \vdash K(\texttt{null}) = \emptyset} \qquad \text{C_LocalVarAcc}\ \frac{}{\Gamma \vdash K(x) = \emptyset}$$

$$\text{C_FieldAcc}\ \frac{}{\Gamma \vdash K(e.f) = \emptyset} \qquad \text{C_MethodAcc}\ \frac{}{\Gamma \vdash K(e.m) = \emptyset}$$

Figure 4.9: Abstract constraint generation for the considered CFG nodes. Γ is the type environment from the type resolution stage, which is a mapping from a node to its qualified type. Since the current stage is after type resolution, the type checker can access the qualified type of every node in Γ, without the need to provide any extra arguments.

For a merge slot α_{merge} that merges α_1 and α_2, we will consider two cases:

$$K(\alpha_{\text{merge}}) = \begin{cases} \{\alpha_{\text{merge}} <: \alpha_1, \alpha_{\text{merge}} <: \alpha_2\} & \text{if } \alpha_{\text{merge}} \text{ is the lower bound of } \alpha_1 \text{ and } \alpha_2 \\ \{\alpha_1 <: \alpha_{\text{merge}}, \alpha_2 <: \alpha_{\text{merge}}\} & \text{if } \alpha_{\text{merge}} \text{ is the upper bound of } \alpha_1 \text{ and } \alpha_2 \end{cases}$$

When α_{merge} is a lower bound, it is a subtype of both operands. When α_{merge} is an upper bound, it is a supertype of both operands. Note that, in the constraints, we do not enforce α_{merge} to be a GLB or a LUB since that part depends on the precision required by a type system. The type system may define mandatory constraints to compute the greatest or the lowest bound, or it may provide soft constraints to prefer a more precise bound.

The generation of these constraints is executed after collecting constraints from other places and before the encoding stage. We can find the slots that are effectively involved in the collected constraints. Then we iterate through the slots to generate the required implicit constraints.

Chapter 5

Discussion and Future Work

Given all the information from previous chapters, this chapter briefly compares UniFlow with the existing frameworks (Section 5.1), discusses the limitations of our framework (Section 5.2), presents our experience in the implementation process (Section 5.3), and explores potential future work (Section 5.4).

5.1 Comparison

This section will compare UniFlow, the Checker Framework and Checker Framework Inference from two angles: (1) functionality and performance, (2) developer and user experience.

5.1.1 Functionality and Performance

Superficially, the three frameworks share some common underpinnings: they are all based on Dataflow Framework [1] and javac's Compiler Tree API[23], i.e., they analyze the same set of nodes and trees. The Checker Framework only supports type checking, whereas the other two frameworks support both type checking and type inference. Checker Framework Inference requires one independent type system with one qualifier hierarchy, while the other two frameworks support multiple type systems with multiple qualifier hierarchies.

The first task of the analysis is type resolution. The Checker Framework executes flow-sensitive type refinement proactively. At each node, it can combine flow-insensitive

[23] https://docs.oracle.com/en/java/javase/17/docs/api/jdk.compiler/com/sun/source/tree/package-summary.html

and flow-sensitive information on demand (i.e., the `getAnnotatedType` method), then it refines the type based on the current context. Although `getAnnotatedType` is a very flexible method, it cannot reuse the types computed before. The main reason is operations on both AST and CFG can control type resolution, so it is difficult to determine if the type of an expression has been completely resolved and refined[24]. A similar discussion has been given in Section 3.1.2, where the types of the operands of a binary expression will be recomputed by the framework. In the worst case, an expression of the form N_1 *op* N_2 *op* ... *op* N_n requires $n(n+1)/2$ type computations. Checker Framework Inference also suffers from the same issue. UniFlow can always reuse the computed types because we know our type environment Γ and stores Σs always contain the latest information from previous nodes, and we only have one source of truth.

The Checker Framework and Checker Framework Inference can use stub files to provide annotations for bytecodes. This functionality is important since annotating commonly used libraries will improve analysis accuracy and reduce false positives. Although the support is not shown in UniFlow's design, we think it is trivial to incorporate stub files by using the existing approach.

In inference mode, both Checker Framework Inference and UniFlow need to traverse ASTs to find appropriate locations to insert solutions. This step visits each tree at most once, so we consider its performance impact as negligible.

The next task in the analysis is applying type rules. In type checking mode, all three frameworks can perform checks against the type rules without using constraint solvers. However, `getAnnotatedType` in the two frameworks is again used commonly in this process to retrieve the types of different expressions, causing performance overhead. In UniFlow, we can reuse the fixed-point results from our dataflow analysis. In type inference mode, there is no significant difference in performance between Checker Framework Inference and UniFlow. Both frameworks need to generate constraints and feed them into some solvers. UniFlow needs some extra resources to compute error messages, but this is necessary and only has a minimal performance impact.

Ultimately, functionality and performance are also type system dependent. Some type systems may have simple lattices with simple program properties to analyze. Some other type systems may have complex, possibly infinite-height, lattices that require special widening and narrowing strategies. Complex lattices and program properties can also complicate the constraints, thus increasing the time on running solvers. We think it will be meaningful to experiment the performance of the same type system implemented on different frameworks.

[24]The only exception as for now is the types on declarations since they are not affected by control flows.

5.1.2 Developer and User Experience

UniFlow is designed to provide a better developer experience than the preceding frameworks. Its two significant contributions are the unified program representation and the unified type rules handling. As discussed in previous sections and chapters, using both AST and CFG to analyze the type of an expression can cause confusion about their objectives or any duplicate logic in code. With the unified program representation, developers know they use CFG for flow-sensitive expressions and AST for flow-insensitive (declaration) constructs. This flattened design allows them to quickly locate and learn the logic for a specific node or tree, thus making the learning curve smoother. On the other hand, Checker Framework Inference proposed that the type rules for type checking can be reused for type inference, but its implementation requires a type system to take different approaches in different modes: use boolean expressions for type checking and generate constraints for type inference. UniFlow unifies this process by only operating on constraints, allowing developers to focus on the actual rules instead of how to handle different modes. Other important improvements are already discussed in Section 3.2.

One advantage of the Checker Framework is it provides various utilities that cover many different aspects, including type defaulting strategies, stub files, patches for javac bugs, etc. These technical details are very helpful for boosting developers' productivity. Another advantage is the Checker Framework manual [32], which thoroughly explains each component and provides instructions for building a new type system. These advantages are the results of many years of development. They are good areas of improvement to UniFlow.

Besides developer experience, we think user experience should also be considered carefully. For analysis behaviour, both the Checker Framework and Checker Framework Inference provide many command line options for users to adjust. Users will have the freedom to decide what assumptions to make (e.g., concurrent semantics assumption [32]) and what files to include or exclude. We also bear these options in mind when implementing our framework. For analysis results, UniFlow can output error messages for inference failure, while Checker Framework Inference only provides the unsatisfiable constraints. There are still many possible improvements to explore. For example, both UniFlow and Checker Framework Inference do not support producing multiple inference solutions. To make the solutions more flexible to users, we may allow users to define some preference rules or even automatically study their preferences from some source code.

5.2 Limitations of the Framework

To avoid having unnecessary complexity in the framework or type system development, we have added several limitations to our framework.

The first limitation is it requires the type system dependencies to have a topological order, i.e., their dependency graph must be a directed acyclic graph. This requirement allows the framework to execute multiple type systems in a consistent order, and each type system does not have to rely on the actual implementation of another type system. The problem with circular dependency is ensuring the same solver applies to multiple type systems. These type systems must have a consensus on the encoding and decoding strategies to make the solver function properly. Based on the existing extensions to the Checker Framework, we have not observed a meaningful use case of mutually dependent type systems. Therefore, the limitation is introduced for better modularity and flexibility.

The second limitation is AST is still essential for providing location and context information, for CFG represents the control flow instead of the syntax structure of the program. In Chapter 4, we have presented the use of AST in processing declarations and finding slot and error locations. However, it is possible that a type system requires context information when analzying a node. An example is already given in Section 4.5.3, where we introduce the helper function currentMethod(). We do not think retrieving such information through the AST is a bad idea, as there is still a clear distinction between AST and CFG. Moreover, the framework can provide many helper functions to retrieve abstract location or context information.

The third limitation is on the power of inference. Many program properties can be too complicated to infer. The question is, to what extent do we want inference to cover? For example, if we want to infer polymorphic qualifiers for a method, we would need to at least generate constraints for each possible combination of its declared types. For a method signature of the form $T_r\ m(T_1\ p_1, \ldots, T_n\ p_n)$, we may have a polymorphic annotation for any of the types $T_r, T_1, \ldots, T_n$. So we need to take $\sum_{i=1}^{n+1} \binom{n+1}{i} = 2^{n+1} - 1$ combinations into consideration. As the size of the program scales up, there will likely be too many constraints for the solver. Due to the complexity, our default constraints in Section 4.5.3 do not infer polymorphic qualifiers. It only guarantees that the existing polymorphic qualifiers are respected. Ultimately, the inference power is determined by the specific type system, which involves more research and discussion.

5.3 Implementation and Challenges

We have implemented a basic version of our framework, which is publicly accessible on GitHub[25]. We tested the framework using a simple type system with a two-element type lattice. We observed expected results from both type checking and type inference, implying our design and ideas are actually feasible. Compared to our eventual goal of laying the groundwork, the implementation is still in an early stage. We plan to continue refining our implementation and conducting more sophisticated experiments.

In this section, we want to share some challenges we have experienced in implementing the initial version:

- **Generic Types Support**: Parametric polymorphism is supported in Java by the introduction of generic types [17]. It is not trivial for a type system to handle generic types since many complex topics are involved: we need to consider how to resolve recursive types such as `<T extends Comparable<T>>`, how to handle the upper bound and the lower bound of a type parameter, how to take care of wildcard types, etc. This is currently an obstacle to our implementation, and we will discuss the relevant future work in the next section.

- **Complexity of Javac**: Javac has a complex pipeline for code compilation, and analysis plugins need to have some understanding of its behaviour to function properly. For example, it is possible to retrieve trees that are not yet analyzed by javac, but those trees are only partially initialized, meaning they do not provide sufficient information for most of the analyses. This is one of the motivations for us to treat location as a detached property of slots, which is explained in Section 4.4.2. Another example is most of the annotations on the declaration bytecode are not directly accessible. This may have a negative impact on the accuracy of computed default types. According to the related workarounds in the Checker Framework, the solution is to process some raw data from the bytecode[26]. We believe that reading the documentation and writing test cases are effective ways to learn the behaviour of javac.

- **Defaulting Strategies Support**: Determining the default qualifier for an unannotated type use is a type system's basic functionality. Although the strategy will be specified by the type system as explained in Section 4.5.2, it is the framework's

[25] https://github.com/zcai1/uniflow

[26] https://github.com/eisop/checker-framework/blob/1b92dc5b55f25c061c963cbb14f0876748972d1d/framework/src/main/java/org/checkerframework/framework/type/TypesIntoElements.java

responsibility to provide the necessary groundwork. From our experience, there are two pieces of information crucial to this process. The first one is the position of the current type relative to the outermost type. For example, `Object` can be a simple `Object` type, but it can also represent the upper bound of a wildcard type, such as `List<? extends Object>`. The second piece of information is the context of the current type use. For example, a type system may want to unsoundly assume the top qualifier as the default for parameters in bytecode, allowing it to loosen type rules for some methods.

5.4 Future Work

The previous sections have covered many functionalities and experiments we will continue working on, including support for stub files and other defaulting strategies. In this section, we want to discuss future topics that need more elaboration.

5.4.1 Generic Types Support

The first question to address for generic types is how to construct a qualified type where a type use appears in its own type declaration or mutually depends on another generic type. An example of the first case is `<T extends Comparable<T>>`, and an example of the second case is `<T extends Comparable<S>, S extends Comparable<T>>`. From our design for data classes in Section 4.3, qualified types are immutable, and thus it is difficult to create a recursive reference among multiple qualified types. Our tentative solution is to use reassignable (i.e., non-final) fields internally in the framework layer, and we can make the property effectively non-reassignable to type systems. More importantly, the framework should carefully handle all recursions to avoid recomputing the qualifier of any constructed qualified types.

Once we can properly handle recursive types, the qualified types will cover all possible `TypeMirror`s from javac, including type variables and wildcard types. We will also update all the type visitors to provide default logic and prevent infinite recursion. This step involves many targets, such as how stores are combined and what constraints to generate for generic types. Luckily, it is possible to obtain some inspiration from the Checker Framework.

The next question we need to think carefully is how to perform type refinement for generic types. Consider the following code snippet:

```
1 List<Cat> cats = new ArrayList<>();
2 List<Animal> animals1 = cats; // invalid
3 List<? extends Animal> animals2 = cats; // valid
```

Line 2 fails to type check because generics are typically invariant in Java, but line 3 passes the type check because the wildcard `? extends Animal` is covariant. There is another form of wildcard `? super T` that is contravariant. Type rules for qualifiers can simply follow these existing rules in Java, but whether to refine the upper or lower bound of a wildcard type remains an open question.

Java also allows writing raw types in the source code, i.e., types with no type arguments specified. This will suppress some type rules to allow unsound operations at compile time. For pluggable type systems, the question is if we want to make conservative assumptions for the ignored type arguments. Additionally, we should consider if it is meaningful to have inference for the type arguments. It is possible that we have an inference solution, but we cannot insert the solution back and pass Java's type system.

5.4.2 Local Type Inference

Local type inference is a big topic that pertains to generic types and type inference. According to Chapter 18 of Java Language Specification [13], local type inference is needed for "generic method applicability", "generic method invocation type inference", and "functional interface parameterization inference". Since pluggable type systems do not affect runtime semantics, it is not their duty to find the applicable generic method. The other two use cases can be demonstrated using the following code:

```
1 List<Cat> cats = new ArrayList<>();
2 Predicate<? super Cat> isGrey = c -> c.getColour() == GREY;
```

Line 1 uses "generic method invocation type inference", where the type argument to the constructor invocation is inferred as `Cat`. Line 2 shows "functional interface parameterization inference", where the parameter type of the lambda expression is inferred as `Cat`.

For consistency, it is beneficial for pluggable type systems to follow the same direction, meaning type qualifiers should obey the same local inference rules. There will be a substantial improvement for analyzing reactive stream operations, which are often written as a chain of generic method invocations with lambda expression arguments.

By following the instructions in the specification, we think it should be straightforward to generate proper constraints for the relevant contexts, and we can merge them into other constraints to feed into the solver. The drawbacks of this solution are: (1) adding these constraints may slow down the solver, and (2) it may not be easy to debug if some constraints are unsatisfiable.

The other option to implement local type inference is by building a dedicated solver, which can compute the solutions after receiving a piece of context information (e.g., a statement). Since the dedicated solver only focuses on a small piece of code and a few types of constraints, it could be more efficient in solving the constraints and easier to debug. However, the amount of required effort needs to be measured carefully.

5.4.3 Bytecode Support

Compared to AST, an advantage of CFG is it can represent the control flow of bytecode instructions. Many previous works have proposed ways to generate CFGs from bytecode to perform static analysis [2, 36]. Since UniFlow is a CFG-based framework, running analysis on bytecode will be an interesting research direction. In addition, we believe allowing type inference on bytecode can provide more precise information for analyzing the source code that depends on it.

There are two foreseeable challenges. First, we need to find ways to retrieve information that we used to get from AST. Fortunately, the element objects in javac will provide information about declarations in bytecode. The difficult part is providing meaningful outputs to type system users, such as the location of a bytecode instruction and the stub files containing inference solutions. Second, since the CFG nodes will represent bytecode instructions, they are very different from the existing nodes generated from AST. We need to have special type resolution and constraint generation strategies for these nodes. For example, an assignment expression in the source code may be translated into a sequence of `load` and `store` instructions. So a valid question would be how to model the stores in regard to memory locations.

Chapter 6

Related Work

6.1 Type System vs. Flow Analysis

Several works have compared type system with control flow (or dataflow) analysis. Palsberg and O'Keefe presented a safety analysis, which is a flow-based analysis for detecting semantics errors in a program, is equivalent to a type system that checks if a program is well-typed [26]. Although the constraints required for type system and for control flow analysis are fundamentally different, they proved the two approaches will accept the same programs.

Heintze not only demonstrated four control flow analyses for type safety but also provided four type systems that deduce control flow properties [15]. This work blurs the distinction between type system and flow analysis since, for certain abstract program properties, they are just two different approaches for the same goal.

While the previous two studies were based on higher-order languages, we have similar results for imperative languages. Laud et al. defined a framework containing some common dataflow analyses for imperative languages, and they showed a general way to produce the equivalent type system for each analysis in the framework [19]. Examples of a few dataflow analysis translations were given in the paper to provide the concrete results obtained by their method.

As concluded by Cousot, type system and program analysis are both a kind of abstract interpretation [9]. The proofs and examples from previous works have established a solid foundation for the feasibility of implementing a type system as a dataflow analysis, thus reinforcing the validity of our work.

6.2 Type Analysis Frameworks

Besides the Checker Framework and Checker Framework Inference, there are other research projects on building frameworks that support pluggable type systems. JavaCOP is a framework developed by Markstrum et al. for defining and verifying type constraints for Java programs [3]. The type constraints (i.e., type rules) are expressed in declarative syntax using their special rule language, which works well with pattern-matching but lacks expressiveness compared with the visitor pattern adopted in the Checker Framework and UniFlow. A follow-up work shows improvements to the framework by adding dataflow analysis and a testing framework for type systems [22].

Greenfieldboyce and Foster proposed JQual [14], which accepts custom type qualifiers and performs type inference for Java programs. The project supports context-sensitive interprocedural analysis by building constraint graphs and solving reachability problems, instead of using constraint solvers. However, there are many limitations: it only supports a single type system with a finite type lattice, it cannot handle generic types, and it is unsound for annotations on the bytecode.

Infer is a static analysis tool that supports C, Objective-C, and Java, and it is now owned by Facebook [8]. Its analysis is based on separation logic [25] and bi-abduction [7], which is claimed to be scalable for large projects. A derivation of the tool, named Infer.AI, has been developed to provide a more general framework with abstract interpretations[27]. Although this is an open source framework with many implemented static analyses (e.g., purity and nullness), it has no publications or documentation about how to implement a custom type system or any implementation details.

Schubert et al. built PhASAR, a dataflow analysis framework for C and C++, which operates on LLVM IR [29]. Similar to JQual, this framework also provides interprocedural analysis by solving graph reachability problems. It allows developers to override transfer functions for different program properties, but few ways exist to encode constraints due to the fixed set of solvers. The paper also mentions that PhASAR is currently soundy but not sound. Overall, we see the potential of implementing an LLVM-based UniFlow by extending this framework, which may be a reasonable future direction.

[27] https://fbinfer.com/docs/about-Infer

6.3 Pluggable Type System Approaches

This section explores the approaches of different pluggable type systems. Using the Checker Framework is a popular and straightforward approach. Crypto Checker by Xing et al. detects improper use of cryptographic algorithms [35]. It uses the Checker Framework to define type lattice and type rules that can flexibly support various forms of cryptographic algorithms. PUnit developed by Xiang et al. provides modular type checking and whole-program type inference for any units of measurement [34]. Checker Framework Inference allows it to build a sound analysis over sophisticated lattices and constraints.

Formal verification involves more complex techniques but also can be more expressive. Lanzinger et al. proposed a way to improve the precision of property type system with the power of formal verification [18]. For property type errors reported by a type system, they suggested adding assertions for the expected properties and using deductive verfication tools to verify those assertions, thus reducing false positives. Infer also uses formal verification thanks to its strong foundation on separation logic [8]. It claimed to have decent precision while running incremental analysis based on bi-abduction.

Machine learning is currently a trending field that also contributes to the development of pluggable type systems. DeepTyper by Hellendoorn et al. performs type inference for TypeScript and JavaScript [16]. The tool features a deep learning model that attempts to place the correct type annotations for a given context, which is achieved by learning from a collection of well annotated programs. TypeWriter is another tool aiming for type inference on dynamically typed languages [28]. It is trained on both the context and the natural language information to provide better understanding of the source code. In addition, it uses a static type checker to rule out the predictions that cannot type check, thus refining the inference results.

Chapter 7

Conclusion

We present UniFlow, a CFG-based framework that helps developers to build precise and expressive pluggable type systems. Based on our past experience in developing other type system frameworks, we carefully design the new framework to provide a better experience for both type system developers and users. With the main focus on dataflow analysis, our framework specifies the tasks on different program representations, making the analysis process more straightforward and more maintainable. There are two important layers in the framework: the framework layer and the type system layer. This design abstracts the underlying details of the framework, allowing developers to focus on writing the code for their analysis. Moreover, our framework requires type rules to be expressed as constraints. Developers will only need to define the type rules for once, and their type systems will support two different modes: a modular type checking mode and a whole-program type inference mode, with consistent behaviours. Ultimately, users can expect helpful error messages with location information in any mode.

Compared to the Checker Framework and Checker Framework Inference, UniFlow provides a smoother learning curve and theoretically some performance gains. The two former frameworks are still more refined due to many years of development. Their abundant documentation and functionalities are great resources for learning. On the other hand, the framework itself has a few limitations on what type systems can do. These limitations will not affect the soundness of the type systems, and we consider them as safe trade-offs for less complexity.

The project is still in its infancy, and many challenges remain to be conquered. We have implemented a prototype with limited functionalities to explore the feasibility of our design, where the initial results are encouraging. We have two important upcoming tasks:

to support generic types and to conduct more comprehensive experiments. For a longer time frame, we believe supporting bytecode analysis will bring significant improvements and new research topics.

References

[1] *A Dataflow Framework for Java*. [Online]. Available: https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf (visited on 01/20/2023).

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost Analysis of Java Bytecode," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07, Braga, Portugal: Springer-Verlag, 2007, pp. 157–172, ISBN: 9783540713142.

[3] C. Andreae, J. Noble, S. Markstrum, and T. Millstein, "A Framework for Implementing Pluggable Type Systems," *SIGPLAN Not.*, vol. 41, no. 10, pp. 57–74, Oct. 2006, ISSN: 0362-1340. DOI: 10.1145/1167515.1167479.

[4] S. Banerjee, L. Clapp, and M. Sridharan, "NullAway: Practical Type-Based Null Safety for Java," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 740–750, ISBN: 9781450355728. DOI: 10.1145/3338906.3338919.

[5] B. A. Becker *et al.*, "Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '19, Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 177–210, ISBN: 9781450375672. DOI: 10.1145/3344429.3372508.

[6] G. Bracha, "Pluggable Type Systems," in *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[7] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional Shape Analysis by Means of Bi-Abduction," *J. ACM*, vol. 58, no. 6, Dec. 2011, ISSN: 0004-5411. DOI: 10.1145/2049697.2049700.

[8] C. Calcagno *et al.*, “Moving Fast with Software Verification,” in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds., Cham: Springer International Publishing, 2015, pp. 3–11, ISBN: 978-3-319-17524-9.

[9] P. Cousot, “Types as Abstract Interpretations,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’97, Paris, France: Association for Computing Machinery, 1997, pp. 316–331, ISBN: 0897918533. DOI: 10.1145/263699.263744.

[10] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller, “Building and Using Pluggable Type-Checkers,” in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 681–690. DOI: 10.1145/1985793.1985889.

[11] W. Dietl, S. Drossopoulou, and P. Müller, “Generic Universe Types,” in *ECOOP 2007 – Object-Oriented Programming*, E. Ernst, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 28–53, ISBN: 978-3-540-73589-2.

[12] W. Dietl, M. D. Ernst, and P. Müller, “Tunable Static Inference for Generic Universe Types,” in *ECOOP 2011–Object-Oriented Programming: 25th European Conference, Lancaster, Uk, July 25-29, 2011 Proceedings 25*, Springer, 2011, pp. 333–357.

[13] J. Gosling *et al.*, *The Java Language Specification, Java SE 17 Edition*. 2021. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se17/html/index.html.

[14] D. Greenfieldboyce and J. S. Foster, “Type Qualifier Inference for Java,” in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA ’07, Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 321–336, ISBN: 9781595937865. DOI: 10.1145/1297027.1297051.

[15] N. Heintze, “Control-Flow Analysis and Type Systems,” en, in *Static Analysis*, A. Mycroft, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1995, pp. 189–206, ISBN: 978-3-540-45050-4. DOI: 10.1007/3-540-60360-3_40.

[16] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, “Deep Learning Type Inference,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 152–162, ISBN: 9781450355735. DOI: 10.1145/3236024.3236051.

[17] A. Igarashi, B. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99, Denver, Colorado, USA: Association for Computing Machinery, 1999, pp. 132–146, ISBN: 1581132387. DOI: 10.1145/320384.320395.

[18] F. Lanzinger, A. Weigl, M. Ulbrich, and W. Dietl, "Scalability and Precision by Combining Expressive Type Systems and Deductive Verification," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485520.

[19] P. Laud, T. Uustalu, and V. Vene, "Type systems equivalent to data-flow analyses for imperative languages," *Theoretical Computer Science*, vol. 364, no. 3, pp. 292–310, Nov. 2006, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.08.013.

[20] J. Li, "A General Pluggable Type Inference Framework and its use for Data-flow Analysis," Master's Thesis, University of Waterloo, 2017. [Online]. Available: http://hdl.handle.net/10012/11771.

[21] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov, "Java Bytecode Verification for @NonNull Types," in *Compiler Construction*, L. Hendren, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 229–244, ISBN: 978-3-540-78791-4.

[22] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble, "JavaCOP: Declarative Pluggable Types for Java," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 2, Feb. 2010, ISSN: 0164-0925. DOI: 10.1145/1667048.1667049.

[23] A. Møller and M. I. Schwartzbach, *Static Program Analysis*, Oct. 2018. [Online]. Available: http://cs.au.dk/%5C~amoeller/spa/.

[24] R. Monat, A. Ouadjaout, and A. Miné, "Static Type Analysis by Abstract Interpretation of Python Programs," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, R. Hirschfeld and T. Pape, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 17:1–17:29, ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.17.

[25] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local Reasoning about Programs That Alter Data Structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01, Berlin, Heidelberg: Springer-Verlag, 2001, pp. 1–19, ISBN: 3540425543.

[26] J. Palsberg and P. O'Keefe, "A Type System Equivalent to Flow Analysis," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 4, pp. 576–599, Jul. 1995, ISSN: 0164-0925. DOI: 10.1145/210184.210187.

[27] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical Pluggable Types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, Jul. 2008, pp. 201–212.

[28] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: Neural Type Prediction with Search-Based Validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 209–220, ISBN: 9781450370431. DOI: 10.1145/3368089.3409715.

[29] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An Inter-procedural Static Analysis Framework for C/C++," en, in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 393–410, ISBN: 978-3-030-17465-1. DOI: 10.1007/978-3-030-17465-1_22.

[30] L. Sun, "An Immutability Type System for Classes and Objects: Improvements, Experiments, and Comparisons," Master's Thesis, University of Waterloo, Apr. 2021. [Online]. Available: http://hdl.handle.net/10012/16882.

[31] M. Ta, "Context Sensitive Typechecking And Inference: Ownership And Immutability," Master's Thesis, University of Waterloo, Apr. 2018. [Online]. Available: http://hdl.handle.net/10012/13185.

[32] *The Checker Framework Manual: Custom pluggable types for Java*. [Online]. Available: https://checkerframework.org/manual/ (visited on 01/20/2023).

[33] G. Van Rossum *et al.*, "Python Programming Language," in *USENIX annual technical conference*, Santa Clara, CA, vol. 41, 2007, pp. 1–36.

[34] T. Xiang, J. Y. Luo, and W. Dietl, "Precise Inference of Expressive Units of Measurement Types," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428210.

[35] W. Xing, Y. Cheng, and W. Dietl, "Ensuring Correct Cryptographic Algorithm and Provider Usage at Compile Time," in *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, ser. FTfJP 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 43–50, ISBN: 9781450385435. DOI: 10.1145/3464971.3468418.

[36] J. Zhao, "Analyzing Control Flow in Java Bytecode," in *16th Conference of Japan Society for Software Science and Technology*, Citeseer, 1999, pp. 313–316.