Pluggable Properties for Program Understanding: Ontic Type Checking and Inference

by

Zhuo Chen

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

Waterloo, Ontario, Canada, 2018

Abstract

Pluggable type systems is a powerful approach to add additional information on types, which can facilitate the understanding of programs. This thesis presents our work on three pluggable type systems for helping both programmers and other program analysis tools to better understand programs.

Pluggable type checking ensures the absence of formalized vulnerabilities. This thesis presents the EOF Value Checker, which prevents unsafe end-of-file (EOF) value comparisons. Unsafe EOF value comparisons can lead infinite loops in programs, and the details are described by the SEI CERT Oracle Coding standard for Java rule FIO-08J. EOF Value Checker examines additional safety-related properties on integer types, and statically ensures no FIO-08J rule violations appear in a given program.

Traditionally, pluggable type inference alleviates manual annotation efforts for type checking. This thesis presents work on extending the purpose of type inference to also produce high-level abstractions for programs. We present a novel type system called Ontology, which reasons about a coarse abstraction for a given program based on ontic concepts. An ontic concept is a high-level semantic conclusion of concrete types or fields in programs. The goal of Ontology is to produce reasonable semantic abstractions for a given program, so that the produced abstraction may facilitate other program analysis tools on their tasks.

To effectively solve type constraints for Ontology, as foundational work, we extend the solver framework in the Checker Framework Inference from only supporting satisfiability problem (SAT) solvers to also supporting Satisfiability Modulo Theories (SMT) solvers. Then, a new SMT encoding approach based on this extension is proposed for Ontology type system. We also apply this new encoding on Dataflow type system, which is previous work on reasoning about concrete run-time types for the components in programs. Our new encoding makes it be possible to support partially annotated programs.

We evaluate EOF Value Checker on 35 real world projects, and it finds 3 defects, 8 bad coding practices, and no false positives are generated. Ontology and Dataflow type systems are evaluated on 15 scientific libraries (range from 393 to 86k LoC). For Ontology type system, it summarizes 4937 built-in ontic concepts, and propagates 274 domain-specific concepts in two pre-annotated physical libraries. For Dataflow type system, the new approach propagates 1.56 times more interesting Dataflow types. In addition, we manually examine the inference results, and verify the two type systems produce meaningful program abstractions of projects in the benchmark. These results suggest that pluggable type systems can provide confidence on preventing formalized vulnerabilities, and be able to infer high-level abstractions for programs.

Acknowledgements

I would like to thank all the little people who made this thesis possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Figures

Chapter 1

Introduction

Program understanding is an important aspect in terms of developing and maintaining software. However, as current software becomes increasingly complex, it is impossible for programmers to thoroughly understand every detail of source code by themselves. Therefore, programmers apply uniform code styles, write code documentations, and design test cases in order to make softwares easier to be understood and less-likely to contain bugs. However, code styles neither help to understand program logic nor on ensuring program to be bug-free. Documentations are easily to be out-of-date. Test cases always ensure programs behave as expected within a specific subset of inputs, and becomes helpless when unexpected inputs occurs.

Type systems can help to enhance program understanding in a formalized way. With assigning types on program constructs, type systems examine the flow of values with types to ensure no type errors happen. Hence, as long as a program passing the type checking by a type system, then it is guarantees this program holds the properties defined by the type rules. The built-in type system in Java is able to prevent many kinds of errors. For example, given below code:

```java
int x = "Hello World!";
```

Java compiler will issue a type incompatible error, as variable `x` has type `int`, but the assignment expression try to assign a `String` type value to `x`. However, the built-in type system in Java has its limitation and cannot prevent every kind of errors. For illustration, Null Pointer Exception is a very common bug and is called "the billion dollar mistake" [9], as it is so hard to detect even if programmers think they code carefully and testing thoroughly. Unfortunately, Java built-in type system cannot prevent Null Pointer Exception. Fig. 1.1 shows a code example that contains a Null Pointer bug, but the Java built-in type system still let it compile.

```
1 void m(Object o) {
2     o.toString();
3 }
4
5 static void main() {
6     m(null);
7 }
```

Figure 1.1: A Code Example for Illustrating Null Pointer Exception

To break out the limitation of Java built-in type system, the Checker Framework [13] has been proposed and implemented. Checker Framework is a tool that using Java Annotation processor to support adding pluggable type systems [7] to the Java language. A type system designer can use the Checker Framework to enforcing additional properties by defining type qualifiers and their semantics. A programmer can then annotate programs with type qualifiers to prevent related bugs. The Checker Framework does not only providing an infrastructure for building additional type systems for Java, but also come with a collection of pluggable type systems that prevent several common kinds of bugs in practice. For example, Nullness Checker in the Checker Framework can be used to prevent the potential Null Pointer Exception in Fig. 1.1. By default, Nullness Checker assume every type use is non-null, and will issue an error on line 6 about passing `null` to a non-null method. Programmers can annotate the method parameter `o` on line 1 with annotation `Nullable` to override this assumption. Then the Nullness Checker will issue an error on line 2 about dereferencing the null-possible reference `o`. Either way, Nullness Checker prevents this Null Pointer Exception.

By applying type qualifiers on types, Checker Framework enrich type semantics and be able to enforce more properties than the Java built-in type system could. However, since programmers have to annotate programs with type qualifiers to be able to use pluggable checkers, manual annotations becomes a heavy burden when annotating legacy libraries or huge and complex programs.

Checker Framework Inference has been presented to alleviate the burden of manual annotations. Instead of performing type checking, Checker Framework Inference generate constraints according to type rules between variables in the program. Then a solver will solve these constraints to give a solution. As long as a solution is found, then this solution is guaranteed to provide a set of annotations annotated to variables in the program to make the program type check. Thus, Checker Framework Inference auto-infer type annotation by this constraint-based approach.

This thesis presents three pluggable type systems built on top of Checker Framework and Checker Framework Inference, to help programmers coding with more confidence and to help other program analysis tools perform better analysis tasks.

We first present EOF Value type system, which prevent FIO-08J rule violation statically. The SEI CERT Oracle Coding Standard for Java rule FIO08-J "Distinguish between characters or bytes read from a stream and -1" describes a rule of how to correctly use Java read APIs. Since the `read` methods in `InputStream` and `Reader` in Java both return an `int` value to represent the read `byte/char` value, it can leading ambiguous reading result if the read `int` value has been converted to `byte/char` before comparing to the end-of-file (EOF) value. This may cause a read-until-EOF-loop either exit prematurely or be stuck in an infinite loop. EOF Value type system ensure every read `int` value compare with EOF value before the narrow-down conversion. Therefore, it effectively prevent this kind of bug. In an evaluation of 35 projects (9 million LOC) EOF Value type system detected 3 defects in production software, 8 bad coding practices, and no false positives.

Then we present the improvement on Dataflow type system [11]. Dataflow type system is an inference type system that concludes the concrete runtime types of each methods may return. In this thesis, a new approach is proposed to encoding and solving constraints for Dataflow type system, which will infer more complete result and makes Dataflow type system possible to support partially annotated programs. We evaluate Dataflow type system in 4 scientific projects. In this case study, our new approach is able to to infer 1.56 times more interesting Dataflow types than the previous approach.

Finally, we present the Ontology type system, which infer and propagate ontic concepts in programs. An ontic concept is a high-level semantic conclusion of concrete types or fields in programs. One straightforward example for illustrating ontic concept is the SEQUENCE concept. SEQUENCE represents a collection of elements organized in some orders. Ontology groups Java `Array` type, `List` and `List` subtypes to SEQUENCE concept. The goal of Ontology is to produce reasonable semantic abstractions for a given program, so that the produced abstraction may facilitate other program analysis tools on their tasks. We evaluate Ontology type system on 15 scientific libraries. The experimental result shows Ontology is able to summarizes 4937 built-in ontic concepts, and propagates 274 domain-specific concepts in two pre-annotated physical libraries. We manually examine some infer results on several random selected projects, and found Ontology is able to infer meaningful results.

1.1 Motivation

The motivation of this thesis is to develop several pluggable type systems that are helpful for programmers and other program analysis tools to better understanding a given program. EOF Value type system is proposed in this thesis, as currently there are no freely available tool for preventing the violation of FIO-08 J rule, and reading is one of the most basic operations in many applications and is required in many different domains. Therefore we want to researching and developing a practical type system that is be able to formally guarantee the absence of this kind of errors from programs.

Code similarity tools often perform code matching based on their control flow graphs and other informations. Categorize types in the graph into higher-level concepts may be helpful for these tools to perform a better similarity task, as now they are performing searching on a higher-level abstraction. Therefore, we propose Dataflow and Ontology type systems in this thesis, to see if they are helpful for these code similarity searching tasks.

1.2 Approach

For EOF Value type system, we designed type qualifiers and type rules, and implement it on top of the Checker Framework. For Dataflow and Ontology type system, we first refactoring and extending the solver framework in Checker Framework Inference from only support SAT back-ends to also support SMT back-end. Then we implement a SMT solver backend based on Z3 for Checker Framework Inference. Next we propose a new encoding based on Set Theory for Dataflow and Ontology type system, and solving the constraints using the SMT solver back-end based on Z3.

For case studies, we evaluate EOF Value Checker on 35 wide-used Apache projects. We evaluate Dataflow and Ontology type systems on a collection of 15 scientific libraries. For the evaluation on each type systems, we collect different aspect of statistics to illustrate how well the type system works.

1.3 Thesis Contributions

This thesis made below contributions:

For type checking, this thesis proposed a new type system for preventing the violation of FIO-08 J rule. To the best of our knowledge, this is the first open source tool that formally prevent this rule violation.

For type inference, we refactoring the Solver Framework in Checker Framework Inference, to let it also support SMT solver back-end. Then we propose a new encoding based on Bit Vector Theory for type systems that theoretically has infinite type qualifiers structured in a power-set-like way, and applying this encoding for Dataflow type system and Ontology type system.

We also propose Ontology type system, which is a novel type system that reasons about high-level abstractions for programs that may facilitate other program analysis tools on their tasks.

In addition, several utility tools are developed during my Master period, that are useful for facilitating research work on projects based on the Checker Framework and the Checker Framework Inference. These tools are not discussed in this thesis in detail, as they are just utility tools. However, it is still worth to mention them for future uses:

- *Test Minimizer*[1]: A tool that helps minimize a test case from a real project, which exposes a bug in Checker Framework or Checker Framework Inference. The minimized test case will expose the same bug as the real project, but will only contain the minimal code necessary to reproduce the bug.
- *corpus-utils*[2]: A Collection of utility scripts for auto-executing tasks for the Checker Framework and Checker Framework Inference on a set of projects (corpus). The tasks include: 1) running CF/CFI on the corpus. 2) check execution log in each project, and delegate to *TestMinimizer* to produce a minimized test case if exceptions happened. 3) collect statistic data and produce latex tables.
- *Checker Framework Live Demo*[3]: A live demo website[4] for the Checker Framework.

[1]See `https://github.com/opprop/do-like-javac/blob/master/README_testminimizer.md`.
[2]See `https://github.com/opprop/corpus-utils/blob/master/README.md`.
[3]See `https://github.com/eisop/webserver/blob/master/README.md`.
[4]See `http://eisop.uwaterloo.ca/live`.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chap. 2 introduces the general background of the Checker Framework, and the Checker Framework Inference. These are the common background for all works in this thesis. Chap. 3 introduces the EOF Value type system in detail. Chap. 4 discusses the improvement work on *TypeConstraintSolver* of extending it to also support SMT solvers, and introduces a new constraint encoding based on Bit Vector theory. Chap. 5 discusses the improvement work on Dataflow type system of applying the new Bit Vector encoding to this system. Chap. 6 introduces the Ontology type system in detail. Finally, Chap. 7 concludes.

Chapter 2

Background and Related Work

This Chapter explains the common background knowledge that readers need to know for reading this thesis. Sec. 2.1 introduces the Checker Framework, and Sec. 2.2 introduces the Checker Framework Inference.

2.1 Background on Checker Framework

This section gives a basic introduction on Checker Framework.

Checker Framework is a framework that supports adding pluggable type systems to the Java language. Pluggable type systems permit more expressive compile-time checking than the built-in type system of a programming language, so that it can guarantee the absence of additional errors. Checker Framework provides a backward-compatible way of expressing *type qualifiers*, by extending the Java 8 annotation system. Checker Framework serves for both programmers and type system designers. Programmers can easily verifies their programs with additional checkers by simply annotating their programs. Type system designers can easily deploy and evaluate their type systems with the flexible declarative and/or procedural mechanisms provided by Checker Framework for defining type qualifiers, qualifier lattice, parametric polymorphism, type rules, and flow-sensitive refinement rules.

Checker Framework is not only an infrastructure for easily developing pluggable type systems, but also has a collections of practical pluggable type checkers built-in within the framework. *Nullness Checker* is the most representative and practical checker in the framework. *Nullness Checker* statically prevents null pointer errors from the program, and it has two main qualifiers:

- `@NonNull` qualifies types that does not includes the `null` value. For example, a variable of type `@NonNull Boolean` is never null, and always has one of the values: `Boolean.TRUE` or `Boolean.FALSE`.
- `@Nullable` qualifies types that includes `null` value. For example, a variable of type `@Nullable Boolean` might be null, and always has one of the values: `Boolean.TRUE`, `Boolean.FALSE`, or `null`.

For minimizing the programmers' annotation efforts, Checker Framework also provide default mechanism to allow type checkers set default qualifiers for each type use location. The `@NonNull` qualifier is the default qualifier in *Nullness Checker* for most type use locations. Therefore, with merely very few annotation efforts, programmers can benefit from the *Nullness Checker* to prevent null pointer errors from their program. Below code example illustrates the programming errors that *Nullness Checker* checks:

```
@Nullable Object obj; // might be null
@NonNull Object nnobj; // never null
  ...
obj.toString()       // checker warning: dereference might
                     //     cause null pointer exception
nnobj = obj;         // checker warning: nnobj may become null
if (nnobj == null)   // checker warning: redundant test
```

For designing and developing a pluggable type system, the Checker Framework supports defining the four main components of an implementation of a pluggable type system:

1. **Type qualifier and hierarchy**. Type qualifier is attached to every occurrence of a type in the program, and further restricts the values that the attached type can represents. The hierarchy describes subtyping relationships among qualified types in the type system. In Checker Framework, type qualifier are defined as Java annotations, and the extended annotation syntax in Checker Framework [13] has been merged into Java 8 as a standrad annotation feature. The hierarchy can be defined declaratively via meta-annotation `@Subtypeof` on the type qualifiers, or be defined procedurally by overriding methods in `QualifierHierarchy` class.
2. **Type introduction rules**. To alleviate manual annotation efforts, type introduction rules specifies default and implicit type qualifiers that should be treated as present on some types and expressions that do not have explicit written qualifiers by programmers. For example, in *Nullness Checker*, every unannotated reference

types are treated as annotated with `@NonNull` in default. In addition, type qualifiers on some types and expressions can be implicitly determined. For example, in *Nullness Checker*, every literal other than `null` should be implicitly annotated with `@NonNull`. Checker Framework provides a declarative mechanism of defining these type introduction rules. The default qualifiers can be defined via meta-annotation `@DefaultQualifier` and `@DefaultQualifierInHierarchy`. The implicit qualifiers can be defined via meta-annotation `ImplicitFor`.

3. **Type rules.** Type rules defines the type system's semantics and yields a type error if a violation happens. These rules defines what the program properties that the type system would check on. For example, in *Nullness Checker*, only `@NonNull` reference types may be dereferenced. Checker Framework has visitor classes that traverse the program's AST to perform type rules checking. The `BaseTypeVisitor` implements the default subtype checking on qualified types. Type system designers can extend the `BaseTypeVisitor` and override methods in the visitor to enforce their specific type rules in the type system.

4. **Interface to the Compiler.** To integrate a pluggable type system into the Java programming language, a Java compiler interface is needed to indicates which annotations are part of the type system, and also to propagate type system specific compiler command-line options. The Checker Framework provides a base checker class called `BaseTypeChecker` that is a Java annotation processor, so that it servers as the compiler interface for type system as a compiler plug-in.

Above just give a brief overview on how the Checker Framework supports on developing type systems, for detail instructions on how to create a new type system on top of the Checker Framework, please see the Checker Framework manual[1]. As another illustration, Chap. 3 also introduces a simple type system built on top of the Checker Framework.

In addition to support the four main components of a type system, the Checker Framework also provides a default flow-sensitive intra-procedural qualifier refinement. The flow-sensitive refinement may refine a qualified type to a more specific type than its declaration type. For example, the *Nullness Checker* will pass below code:

```
void m(@Nullable Object obj) {
  if (obj != null) {
    obj.toString(); // OK, flow-sensitive refinement
                    // refines obj to @NonNull type
```

[1]See https://checkerframework.org/manual/

```
  }
}
```

The Checker Framework use the Dataflow Framework to implements the default flow-sensitive refinement analysis. Type system designers may optionally overrides the default implementation to perform type system specific flow-sensitive refinement. The Dataflow Framework manual[2] is a good reference material of learning how to extend the default flow-sensitive refinement in the Checker Framework.

2.2 Background on Checker Framework Inference

Checker Framework Inference is proposed to alleviate the manual annotation efforts for the type checkers in the Checker Framework. Although the Checker Framework provides flow-sensitive refinement for reducing the manual annotation efforts, the flow-sensitive refinement is intra-procedure and therefore cannot compute qualified types for global flow analysis. Therefore, Checker Framework Inference is presented to achieve a global whole program type inference.

The approach Checker Framework Inference uses for whole program type inference is a constraint-based strategy. For each program location, Checker Framework Inference generates a *constraint variable* for this location. A *constraint variable* is a placeholder that represents a type qualifier in the type hierarchy for a given program location, and it can be either variable that needs to be solved, or a constant that introduced by type introduction rules in the type system. Then Checker Framework Inference generating *type constraints* among those *constraint variable* based on type rules in the type system. [11] gives a detail description on the available kinds of *type constraints* in the Checker Framework Inference. Finally, Checker Framework Inference will pass these *type constraints* to a generic solver framework introduced by [11], which will encode these *type constraints* to low-level (e.g. MAXSAT) constraints and let a concrete solver solve to produce a solution. Since the *type constraints* are generated by type rules in the type system, the solution is guaranteed to produce a type-checked annotated program. In contrast,no solution can be found would suggests type rule violations exist in the program.

As Checker Framework Inference depends on underlying solvers to solve constraints, the higher-level solvers would support more complex *type constraints*. Chap. 4 describes how we extend the original solver framework in [11] from supporting SAT-solver only to also supporting SMT-solvers.

[2]See https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf

Checker Framework Inference reused the main architecture in the Checker Framework, and build a type inference system on top of the Checker Framework Inference is very similar to build a type system on top of the Checker Framework as described in Sec. 2.1. The ultimate goal of Checker Framework Inference is to substitute the Checker Framework, so that every type system build on top of the Checker Framework Inference can have both type-checking mode for enforcing type rules, and type-inference mode for inferring type qualifiers. Also, Checker Framework Inference also extend the purpose of type system from providing formalized guarantees by type checking to also building higher-level semantic abstractions of a program by type inference. Chap. 5 and Chap. 6 introduce two type inference systems build on top of the Checker Framework Inference, which are aiming for building abstractions of a given program for enhancing program understanding for both programmers and other program analysis tools.

Chapter 3

EOF Value Type System

3.1 Introduction

Reading from an input stream is one of the most basic operations for an application and required in many different domains. Java provides methods `InputStream.read()` and `Reader.read()` for reading bytes or characters from an input stream. `InputStream.read()`[1] returns an `int` in the range of 0 to 255, or -1 if the end-of-file (EOF) was reached. Similarly, `Reader.read()`[2] returns an `int` in the range of 0 to 65535, or -1 for EOF. These `read` methods return an `int` in order to distinguish the additional -1 from the maximum `byte`/`char` value. As the CERT FIO08-J [8] rule describes, one common usage mistake of these `read` methods is the premature conversion of the read `int` to `byte`/`char`, before comparing with -1. In Java, `byte` is defined as an 8 bit signed number, `char` is a 16 bit unsigned Unicode character, and `int` is a 32 bit signed number. The narrowing conversion from the returned `int` to `byte`/`char` makes it impossible to distinguish the maximum `byte`/`char` value from the EOF value -1. Fig. 3.1 shows a simple example of this kind of mistake. If the `int` returned by `Reader.read()` is prematurely converted to a `char`, the EOF value -1 is converted to 65535, resulting in an infinite loop. Similarly, if the `int` returned by `InputStream.read()` is prematurely converted to a `byte`, the maximum stream value 255 is converted to -1, resulting in the loop to exit prematurely.

This Chapter describes the EOF Value Checker, a tool that allows a conversion from the read `int` to `byte`/`char` only after comparing with -1, to guarantee the absence of ambiguously converted results. The tool is designed as a pluggable type system for Java

[1]See https://docs.oracle.com/javase/9/docs/api/java/io/InputStream.html#read--.

[2]See https://docs.oracle.com/javase/9/docs/api/java/io/Reader.html#read--.

```
StringBuffer stringBuffer = new StringBuffer();
char c;
while ((c = (char) reader.read()) != -1) {
    stringBuffer.append(c);
}
```

Figure 3.1: An example of a FIO08-J rule violation, resulting in an infinite loop.

and built using the Checker Framework [7]. With the rich type rules and a standard data-flow framework provided by the Checker Framework, this tool is implemented easily through 312 lines of Java code. This tool guarantees that the FIO08-J rule is never violated. In an evaluation of the tool on 35 real world projects (9 million LOC), it found 3 defects and 8 bad coding styles that violate the FIO08-J rule, and there were zero false positives. Only 47 manual annotations were required in this evaluation. To the best of our knowledge, the EOF Value Checker is the first open source tool that prevents this vulnerability. It is available freely on GitHub[3].

The rest of this chapter is organized as follows. Sec. 3.2 presents the EOF Value Checker type system, Sec. 3.3 presents the implementation of this type system, Sec. 3.4 presents the case study of applying the EOF Value Checker to 35 open source projects, and Sec. 3.5 reviews related work. Finally, Sec. 3.7 concludes.

3.2 Type System

This section presents a qualifier-based refinement type system that guarantees that premature conversions from read `int` to `byte`/`char` never happen at run time. Sec. 3.2.1 introduces the qualifiers and the qualifier hierarchy; Sec. 3.2.2 explains the type rules; Sec. 3.2.3 explains default qualifiers; and Sec. 3.2.4 explains data-flow-sensitive qualifier refinement.

3.2.1 Type Qualifiers and Qualifier Hierarchy

The EOF Value Checker type system provides three qualifiers: `@UnsafeRead`, `@UnknownSafety`, and `@SafeRead`:

[3]See https://github.com/opprop/ReadChecker

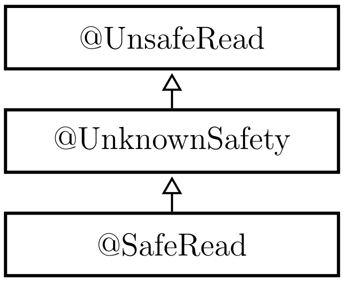


Figure 3.2: Qualifier hierarchy of the EOF Value Checker.

- `@UnsafeRead` qualifies `int` types that represent a `byte/char` or the EOF value -1 **before** being checked against -1.
- `@SafeRead` qualifies `int` types that represent a `byte/char` **after** being checked against -1.
- `@UnknownSafety` qualifies `int` types without any compile time information about their representations.

All three qualifiers are only meaningful for `int` types. All other types can essentially ignore qualifiers.

The type qualifiers form a simple subtype hierarchy. Fig. 3.2 illustrates the subtying among type qualifiers.

It is counter intuitive that `@UnknownSafety` is not the top qualifier, but is a sub-qualifier of `@SafeRead`. The reason of designing hierarchy in this way will be explained in Sec. 3.2.3.

3.2.2 Type Casting Rules

The EOF Value Checker restricts the standard type rules for narrowing casts, as shown in Fig. 3.3. Casts from `@UnsafeRead` to `byte/char` are forbidden and only `@UnknownSafety int` and `@SafeRead int` can be cast to `byte/char`. With the return type of the `read` methods annotated with `@UnsafeRead`, the type cast rules ensure that the read `int` is compared against -1 before being cast to `byte/char`.

These type cast rules will effectively prevent the read mistake by preventing premature type casts. Fig. 3.4 shows the type cast error issued by the EOF Value Checker for the example shown in Fig. 3.1.

$$\frac{\Gamma \vdash e : Q\ \texttt{int} \quad Q \neq \texttt{@UnsafeRead}}{\Gamma \vdash \texttt{(byte)}\ e : \ \texttt{byte}} \qquad \frac{\Gamma \vdash e : Q\ \texttt{int} \quad Q \neq \texttt{@UnsafeRead}}{\Gamma \vdash \texttt{(char)}\ e : \ \texttt{char}}$$

Figure 3.3: Type rules for narrowing casts. Only casts from `@UnknownSafety` and `@SafeRead` are allowed. Casts from `@UnsafeRead` are forbidden. All other type rules are standard for a pluggable type system and enforce subtype consistency between types.

```
error: @UnsafeRead int should not be casted to char.
line 3: while ((c = (char) reader.read()) != -1) {
                    ^
```

Figure 3.4: The type cast error issued for the example from Fig. 3.1.

This type error can be fixed by comparing the read `int` with the EOF value before casting it to `char`. Fig. 3.5 shows the corrected source code.

Note that the cast to `char` is allowed after the comparison against -1. The data-flow refinement, explained in Sec. 3.2.4, refines the type of `data` from `@UnsafeRead` to `@SafeRead` after the -1 comparison, allowing the cast in the loop body. This fixes the premature conversion without requiring any explicit annotations in the source code.

3.2.3 Default Qualifiers

The type system uses default qualifiers for all type uses, minimizing the manual annotation effort. Defaulting follows the CLIMB-to-top approach from the Checker Framework[4]. Local variables are defaulted with the top qualifier, `@UnsafeRead`, because their effective type will be determined with data-flow-sensitive type refinement.

[4] https://checkerframework.org/manual/#climb-to-top

```
StringBuffer stringBuffer = new StringBuffer();
int data;
while ((data = reader.read()) != -1) {
    stringBuffer.append((char) data);
}
```

Figure 3.5: A fix of the type cast error shown in Fig. 3.1.

`@UnknownSafety` is the default qualifier for all other type use locations. Since `@UnknownSafety` is a subtype of `@UnsafeRead`, this means an `@UnsafeRead int` cannot be assigned to a field or passed to a method without explicit `@UnsafeRead` annotation on the field/method parameter. This ensures that an `@UnsafeRead int` isn't lost through a non-local data flow. This is also the reason of designing `@UnknownSafety` as a sub-qualifier of `@SafeRead`, instead of being the top qualifier in the hierarchy. With this design, in default, an `@UnsafeRead int` cannot be assigned to a field or a method parameter according to the qualifier hierarchy.

Our case study finds that most read `int` are used locally. In very few cases programs assign a read `int` to a field or pass them as a method parameter. In the evaluation of 35 projects, only 1 project requires annotating 2 fields and 1 method parameter with `@UnsafeRead`.

3.2.4 Data-flow-sensitive Type Refinement

The EOF Value Checker performs data-flow-sensitive type refinement to minimize the annotation effort. An `@UnsafeRead int` can be refined to `@SafeRead` if the possible run-time values of this `int` are guaranteed to not include -1. Correct programs use range checks or comparisons against -1 to ensure a conversion to `byte`/`char` is safe. These value comparisons provide static information which the EOF Value Checker can use to ensure casts are safe. The EOF Value Checker applies additional transfer functions on binary comparison nodes in the control-flow graph to refine types.

For a binary comparison node, if one of the operands is `@UnsafeRead int` and the other operand is a constant value, the corresponding transfer function refines the `@UnsafeRead int` to `@SafeRead` in the branch that ensures -1 is not a possible run-time value of the `@UnsafeRead int`. Fig. 3.6 gives several examples of the data-flow-sensitive refinement of binary comparison nodes in the EOF Value Checker.

The EOF Value Checker does not perform any constant propagation and only comparisons between `@UnsafeRead int` and literals are refined. This can cause a false positive if an `@UnsafeRead int` is compared with a variable, for which a constant propagation could determine a value. Fig. 3.7 gives an example of this kind of false positive. This could easily be improved by also applying a constant propagation. However, in the evaluation on 35 projects, no false positives are generated, and therefore there are no cases where a constant propagation would help.

```
@UnsafeRead int data = in.read();

// Explicitly compare read result with EOF value -1.
if (data != -1) { /* refines to @SafeRead */ }

// Only non-EOF values can flow into the block.
if (data == '<' || data == '>') { /* refines to @SafeRead */ }

// A range check which excludes the EOF value.
if (data >= 0) { /* refines to @SafeRead */ }
```

Figure 3.6: Some examples of data-flow-sensitive refinement.

```
@UnsafeRead int data = in.read();
final int MINUS1 = -1;

// Transfer functions would not refine data to @SafeRead,
// as MINUS1 is not a literal.
if (data != MINUS1) {
    char c = (char) data; // A false positive warning.
}
```

Figure 3.7: A possible false positive warning due to the absence of constant propagation.

3.3 Implementation

The EOF Value Checker type system described in Sec. 3.2 is implemented as a pluggable type system using the Checker Framework [7].

The type system is independent of the specific stream API. It can be instantiated by annotating methods that need protection against premature conversion as returning `@UnsafeRead`. The EOF Value Checker provides 32 `@UnsafeRead` annotations for the `read` methods in the `InputStream` and `Reader` classes and their subclasses in `java.io`, `java.net`, `javax.swing`, `javax.sound.sampled`, `javax.imageio`, `java.util.zip`, and `java.security` packages. It is easy to provide additional annotations for other APIs.

Overall the implementation effort is very low, totaling 312 lines of Java code. The EOF Value Checker uses the standard type rules and data-flow-sensitive type refinement from the Checker Framework. Only the type rule for casts has been extended as shown in Fig. 3.3. Only three transfer functions on binary comparison nodes are extended to achieve the data-flow-sensitive type refinement described in Sec. 3.2.4.

3.4 Experiments

We evaluate the EOF Value Checker on 35 open source projects. The largest project is Apache TomEE, a lightweight JavaEE Application server framework. The other 34 projects are from Apache Commons, a collection of reusable components in wide use. For each project, the EOF Value Checker is ran on the Java source files with a configuration extracted from the project build file. Fig. 3.8 presents the experimental results.

For every resulting warning, we manually identify whether it is a real defect, a bad coding practice, or a false positive. A real defect is to use the prematurely converted result in a comparison to the EOF value, which might lead the reading loop to exit prematurely or to be stuck in an infinite loop. We categorized a warning as a bad coding practice if the prematurely converted `int` is used before the `int` is compared to the EOF value, which can lead to invalid output.

Overall, the EOF Value Checker finds 3 defects in Apache TomEE and Apache Commons IO and 8 bad coding practices in 3 projects. No false positives are generated for all 35 projects. The 3 defects have been reported to the respective project maintainers. The two issues in Apache TomEE have since been fixed.

Project	Java LOC	Manual Annotations	Bad Style	Defects	Time Overhead
Apache TomEE	1178k	2	2	2	2.9
Apache Commons IO	42k	12	0	1	4.4
Apache Commons BCEL	366k	1	4	0	2.5
Apache Commons Imaging	49k	6	2	0	3.9
Apache Commons Compress	57k	15	0	0	2.9
Apache Commons CSV	9k	2	0	0	1.9
Apache Commons Fileupload	8k	1	0	0	0.9
Apache Commons Net	34k	4	0	0	1.9
Apache Commons VFS	39k	4	0	0	3.6

Figure 3.8: Case study results. Only projects that have defects, bad coding practices, or explicit annotations are listed. The time overhead is relative to the original compile time.

The overall annotation effort is very low. Overrides for the `read` methods need to be annotated, requiring a total of 44 annotations. Only 1 project needs additional `@UnsafeRead` annotations on 2 fields and 1 method parameter.

Running the EOF Value Checker adds compile time overhead. For the largest project, the EOF Value Checker adds 2.9 times the original compile time as overhead. On average 2.75 times overhead is added. This overhead is expected for a Checker Framework based type system. Future performance improvements to the Checker Framework will also benefit the EOF Value Checker.

3.5 Related Work

The Parasoft Jtest PB.LOGIC.CRRV checker is the only existing tool listed on the CERT website for the FIO08-J rule [8]. However, this commercial product was not available to us for evaluation.

FindBugs/SpotBugs[5] has a bug rule “RR: Method ignores results of InputStream.read()” that ensures that methods check the return value of variants of `InputStream.read()` that return the number of bytes read. The case for FIO08-J is not covered. SpotBugs also has a bug rule ”INT: Bad comparison of non-negative value with negative constant or zero” that prevents the comparison of prematurely converted `char`s to the EOF value -1, as unsigned `char`s should not be compared to a negative value. However, neither of these rules prevents the premature conversions and the resulting defects and bad coding practices.

None of the rules in PMD, CheckStyle, and Coverity prevent the premature conversions.

3.6 Future Work

One interesting future work is to generalize the EOF Value Checker so that our type system is not specific to Read APIs, and can prevent all unsafe type cast from a wider numeric type to a narrower numeric type. CERT also has a more general rule NUM12-J[5] which gives a instruction on how to ensure data is not lost or be misinterpreted when converting from numeric types to narrower types.

One possible idea for generalizing EOF Value Checker would be tracking the possible runtime value range of variables of numeric types, and only allow their conversion to narrower types when the possible runtime value range is within the range that narrower types can express. For example, an `int` can only be cast to a `char` when its possible runtime value range is within 0 to 65535. Further more, APIs that return numeric types that is wider than the concrete types of returned data (e.g. Read APIs return `int` to represent `byte` or `char`) should be annotated by the possible value range they may return. For example, read APIs in Java would be annotated with int range from -1 to 255 or 65535 respectively. This range annotation on APIs would allow common coding patterns that programmers use for making sure the conversions of the returned results from these APIs are still allowed in the new generalized type system.

3.7 Conclusions

This chapter presents a qualifier-based type system that guarantees that a premature conversion from a read `int` to `byte`/`char` never happens at run time. We instantiated this type system for Java's read API: `InputStream.read()`, `Reader.read()`, and their overrides in the JDK. We built an implementation on top of the Checker Framework, which only required extending one type rule and three transfer functions in the framework. This implementation is available at https://github.com/opprop/ReadChecker. With only a very low annotation burden, this tool found 3 defects, 8 bad coding practices, and generated no false positives in 35 large, well-maintained, open source projects.

[5]See https://wiki.sei.cmu.edu/confluence/display/java/NUM12-J.+Ensure+conversions+of+numeric+types+to+narrower+types+do+not+result+in+lost+or+misinterpreted+data.

Chapter 4

Type Constraint SMT Solver

Type Constraint Solver is a solver framework introduced by previous master thesis [11]. The purpose of *Type Constraint Solver* is to solve the *type constraints* generated by inference type systems built on top of the Checker Framework Inference. This Chapter discusses the improvement on the *Type Constraint Solver* from supporting Max-SAT-solvers only to also supporting Max-SMT-solvers. This extension is a necessary infrastructure for applying Bit Vector theory on encoding the *type constraints* and solving them in Max-SMT-solvers for Dataflow and Ontology type systems. Details of encoding in Bit Vector theory will be discussed in Sec. 4.3.

The rest of this Chapter is organized as follows. Sec. 4.1 gives a brief introduction on SAT (propositional satisfiability problems) and SMT (Satisfiability Modulo theories). Sec. 4.2 motivates the reason of extending *Type Constraint Solver* from supporting Max-SAT-solvers to also supporting Max-SMT-solvers, and introduces the engineering work on extending *Type Constraint Solver* from SAT to SMT. Sec. 4.3 introduces the first SMT solver back end for *Type Constraint Solver* by integrating the Z3 solver, and discuss the general encoding with Bit Vector theory. Finally, Sec. 4.4 concludes.

4.1 Theory Background

4.1.1 Background on SAT and Max-SAT

SAT problem [1], also called Boolean Satisfiability problem, is the problem of determining if a given boolean formula exists a *satisfiable* solution. A boolean formula can be defined by the following inductive process:

1. an atomic formula is consisted of a single boolean variable.
2. all atomic formulas are boolean formula.
3. For every boolean formula F, the negation $\neg F$ is a boolean formula.
4. For all boolean formula F and G, $(F \vee G)$ and $(F \wedge G)$ are also boolean formula.

A *satisfiable* solution of a boolean formula is a map that assigns each boolean variable in the boolean formula to a boolean value, so that the boolean formula would be evaluated to true. If no such a solution can be found for a boolean formula, we say this boolean formula is *unsatisfiable*. For example, given below boolean formula:

$$(x \vee y) \vee (x \wedge y) \tag{4.1}$$

A satisfiable solution for this formula could be assigning boolean variables x and y all to true. As an example for unsatisfiable boolean formula, contradictions are always *unsatisfiable*.

Sometimes for an unsatisfiable boolean formula, determine a solution that can "maximum" satisfies the formula can be very useful. Maximum satisfiability problem (Max-SAT) [3] is proposed to fulfill this desire. Max-SAT is a generalization of SAT problem, and its goal is to determine the maximum number of clauses of a given boolean formula in conjunctive normal form (CNF) [2] that can be made true by a solution of the boolean variables in the formula. For example, given below boolean formula in CNF form:

$$(x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \tag{4.2}$$

This formula is unsatisfiable, as it is always false regardless the boolean value assignments to its boolean variables x and y. However, it is possible to make at most three clauses of the formula to be true. Therefore, A Max-SAT solution of this formula would be a solution that be able to make three clauses of the formula to be true. More advanced Max-SAT problem categories clauses to *hard clause* and *soft clause*. For a given boolean formula, its goal is to find a solution that satisfies all *hard clause*s, and also maximize the number of *soft clause*s to be true.

4.1.2 Background on SMT and Max-SMT

Satisfiability modulo theories (SMT) problem [4] is a further generalization on SAT problem. SMT is the problem of determining a *satisfiable* solution for a logical formula with respect to combinations of background theories expressed in first-order logic. A logical formula in SMT problem can be defined similarly by the inductive definition in Sec. 4.1.1, with only one difference:

An atomic logical formula in SMT problem is consisted of a single boolean predicate.

A boolean predicate is a binary-valued function of non-binary values, and the values of a boolean predicate is classified according to a corresponding background theory. Common background theories are the theory of integers, the theory of real numbers, the theory of bit vectors, etc. For example, given below logical formula with theory of integers:

$$(x < 0) \wedge (x \leq y) \tag{4.3}$$

The background theory of integer would classify the predicate $(x < 0)$ and $(x \leq y)$ in the formula to boolean values based on the values in the integer theory assigned to the symbol x and y. A satisfiable solution of this logical formula would be any solutions that assign proper values to x and y, so that the boolean formula consisted of the predicates can be evaluated to true. For example, Satisfiable solutions for this logical formula would be those that assigns a negative integer value to x, and assign a value to y that is bigger or equal to the value assigned to x.

Similarly, Max-SMT problem is a generalization of SMT, with the goal of determining a solution that can maximize the numbers of clauses of predicates to be true in the given logical formula in CNF form. Max-SMT problem also can categorized clauses into *hard clause* and *soft clause*, and to find a solution satisfies all *hard clause*s while try to maximizes the number of *soft clause*s to be true.

4.2 Extend Type Constraint Solver to SMT

Type Constraint Solver is a solver framework introduced by a previous Master thesis [11]. The purpose of this solver framework is to solve type constraints generated by the Checker Framework Inference. As a general solver framework, *Type Constraint Solver* is designed to be both type systems agnostic and concrete solvers agnostic. In other words, it is independent to specific type systems, and also can be easily extended by adding more concrete solvers (e.g. sat4j, LogiQL) as back ends.

This section introduces the work on extending the *Type Constraint Solver* from only support SAT solvers to also support SMT solvers. Sec. 4.2.1 motivates why we are interested in extending the solver framework to support SMT solvers. Sec. 4.2.2 gives a brief overview on previous solver framework architecture, and illustrate why this architecture only support SAT solvers. Sec. 4.2.3 introduces the new architecture that support both SAT and SMT solvers.

4.2.1 Motivation

For solving type constraints generated by the Checker Framework Inference, the very first step is to encode these type constraints as low-level constraints that concrete solvers be able to understand. For example, for Max-SAT solvers, the solver framework need to encode the type constraints to a Max-SAT problem, so that the concrete Max-SAT solvers (e.g. sat4j, lingeling) can solve it.

The approach of encoding type constraints to a Max-SAT problem was first proposed by the Generic Universe Type paper [20], and was implemented in the *Type Constraint Solver* [11]. The basic idea of this Max-SAT encoding is an enumeration-based approach. Given a type hierarchy that has n qualifiers, each constraint variable needs n boolean variables to enumerate all qualifiers in the type hierarchy as possible solutions to this constraint variable. For example, given a type hierarchy consisted of two qualifiers TOP and $BOTTOM$, each constraint variable v needs two boolean variables β_v^{TOP} and β_v^{BOTTOM} with below well-form encoding:

$$(\neg\beta_v^{TOP} \vee \neg\beta_v^{BOTTOM}) \wedge (\beta_v^{TOP} \vee \beta_v^{BOTTOM}) \tag{4.4}$$

In above well-form encoding, each boolean variable $\beta_v^{q_i}$ represents the truth assignment of assigning type qualifier q_i to constraint variable v. This well-form encoding ensures the solution will only assign each constraint variable with exactly one qualifier in the type hierarchy.

As the example shows, given a type hierarchy with n qualifiers, for each constraint variable, Max-SAT encoding would encode n boolean variables to enumerate all possible qualifier assignments on this constraint variable. Since Max-SAT is a NP-hard problem, the solving time grows exponentially with the increasing number of boolean variables in the given boolean formula. For some type systems, e.g. Dataflow type system [11], the number of qualifiers in the hierarchy depends on the concrete program that the type system running on, and may have a huge amount (>100) of qualifiers. For these type systems, it is impossible to encoding all type constraints once and let a single SAT-solver instance solve it in a reasonable time. To reduce the solving time, The previous master thesis [11] for

Dataflow proposes a divide-and-conquer approach that separates the constraints set and dispatches subsets of constraints to multiple SAT-solver instances to solving constraints in parallel. However, this divide-and-conquer approach only work for Dataflow type system on unannotated programs, and cannot give a sound inference result for Ontology type system. The details of the limitation of divide-and-conquer approach will be discussed in Chap. 5.

Another possible solution to reduce solving time is to encode type constraints as a SMT problem and then use a SMT solver to solve it. The main benefit of SMT encoding is for some cases, type constraints can be encoded with a significantly fewer number of boolean bits as boolean predicates in a SMT problem than the number of boolean variables needed to encode as a SAT problem. Consequently, the boolean search space for the underlying SMT solver is significantly reduced, so that the encoded SMT problem can be solved in a reasonable time. During this research, we found that for type systems with a power-set-like type qualifier hierarchy (will be discussed in Sec. 4.3.1) that has 2^n qualifiers, each constraint variable in their type constraints can be encoded with n bits as a boolean predicate classified by Bit Vector theory in a SMT problem, instead of 2^n boolean variables in a SAT problem. Consequently, for these type systems, a reasonable solving time can be achieved by encoding their type constraints as a SMT problem with Bit Vector theory.

4.2.2 Previous Architecture and its Limitations

Previously, Type Constraint Solver mainly composed by three components. *Front End* takes the type constraints generated by Checker Framework Inference, and performs some necessary initialization steps of initializing corresponding Back End and Serializer. *Back End* is the adapter that coordinating the solving process with the concrete solver. Back End will first delegate the encoding job to *Serializer*, which would encode type constraints into a Max-SAT formula. Then Back End will pass the encoded formula to the concrete solver, get the solution back, and then decode it to a map from slot's id to the qualifier (the solution of the corresponding slot). Fig. 4.1 shows the previous architecture of type constraint solver.

Notice that the encoding responsibility is taken by *Serailizer*, but the decoding responsibility is taken by *Back End*. Based on this architectural design, both *Serializer* and *Back End* have to know the details of encoded types of *Slot* (the type of constraint variable) and *Constraint* to perform the encoding and decoding tasks. Fig. 4.2 shows the corresponding class diagram of previous Type Constraint Solver.

As both Back End and Serializer aware of the detail encoded types of slot and constraint,

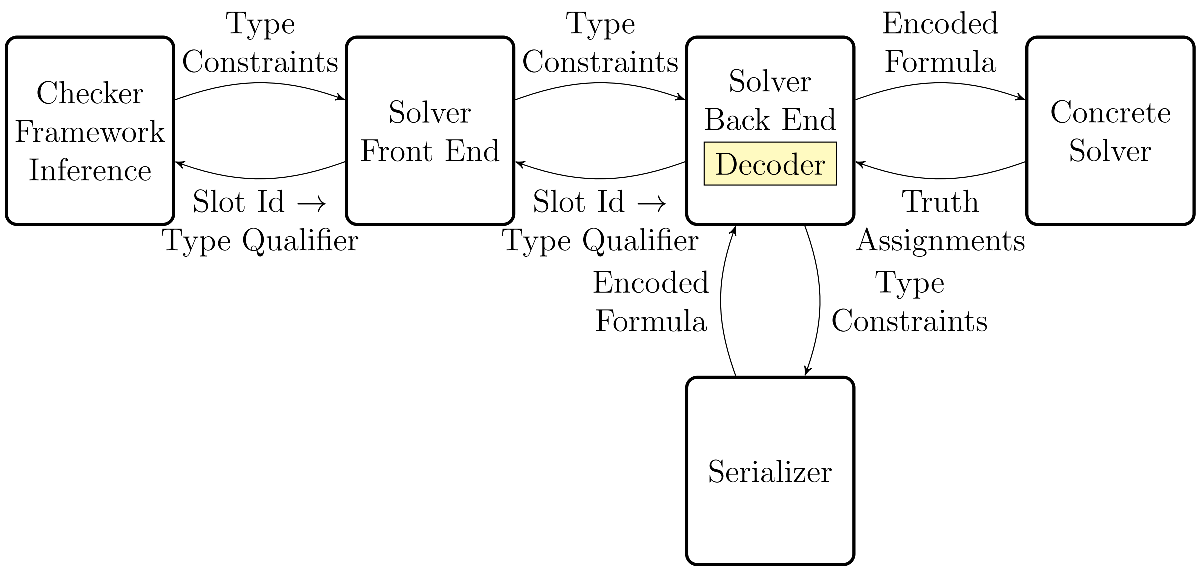


Figure 4.1: Previous architecture of type constraint solver. Notice that the decoder is encapsulated in the solver back end.

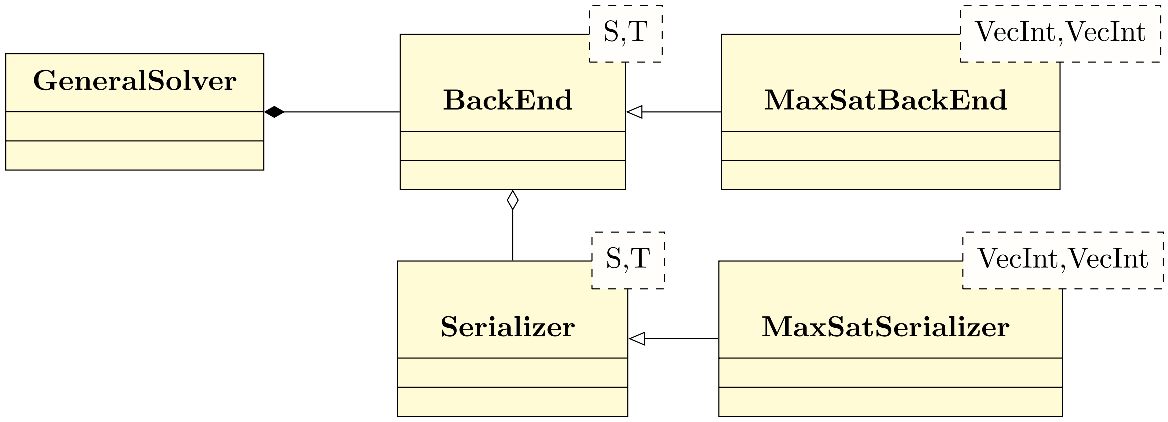


Figure 4.2: The class diagram of previous type constraint solver. The dashed label on classes represent the class type parameters. The type parameter S represents the encoded type of slot, and T represents the encoded type of constraint. Notice that *BackEnd* requires S and T.

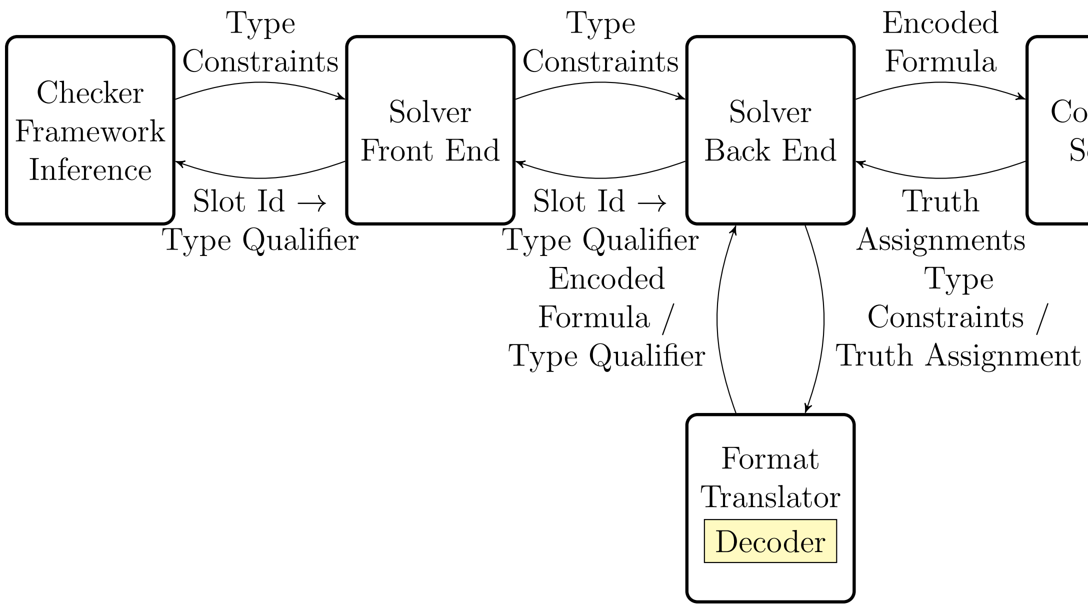


Figure 4.3: The new architecture of type constraint solver. Notice that now format translator takes the responsibility of decoding.

this actually limits one Back End have to coupling with exactly one kind of encoding. For Max-SAT Back Ends, this coupling does not have side-effects as Max-SAT Back Ends always require only SAT encoding. However, when it comes to SMT Back Ends, as the encoding is actually depends on the background theory, the previous architecture design of type constraint solver limits the possibility of having one SMT Back End with the flexibility of choosing the background theory for encoding.

4.2.3 The New Architecture

To resolve the limitations in previous architecture, the new design decoupled encoding with Back End. Specifically, the new design let Serializer takes the responsibility of both encoding and decoding, and Back End only indicates the compatible Serializer type, instead of indicates the detailed encoded type of Slot and Constraint. Fig. 4.3 shows the new architecture.

Noticed in the new architecture, Back End does not takes the responsibility of decoding solution get from concrete solver anymore. Instead, when concrete solver returns solution to Back End, Back End will ask the Format Translator to decode the solution as type

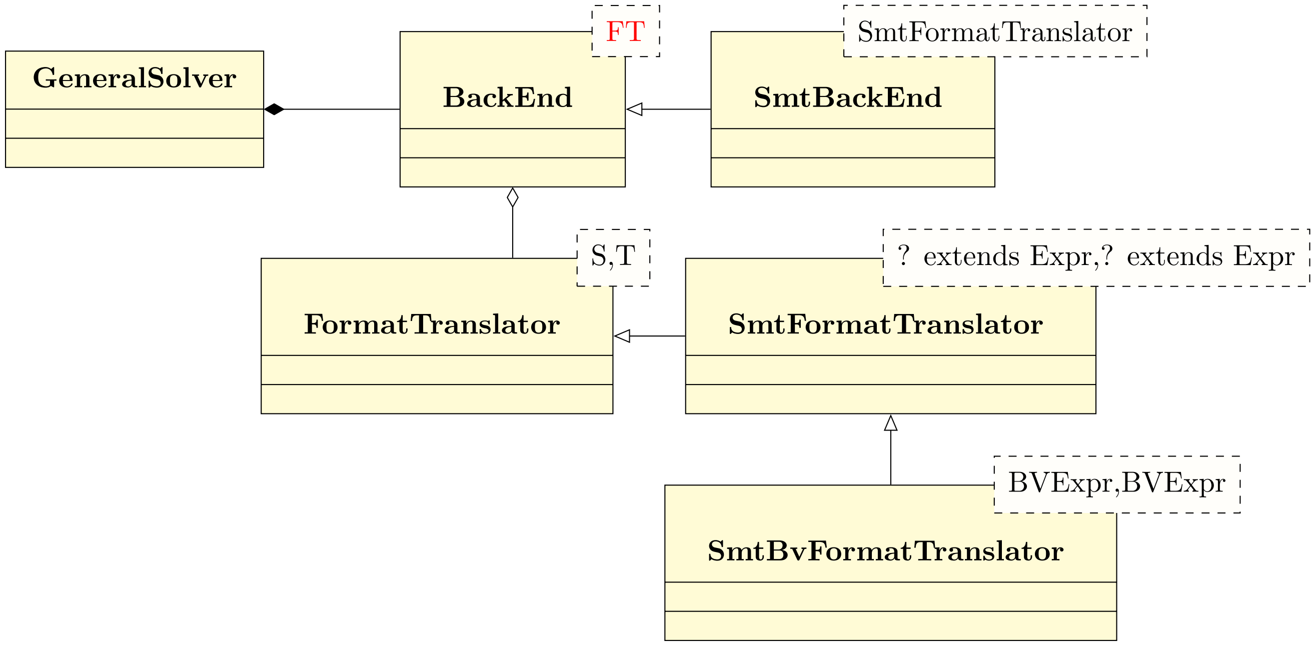


Figure 4.4: The class diagram of the new type constraint solver. Notice that now back end only requires a type parameter FT, that represents the type of format translator for this back end. SMT back ends can require a general type of abstract SMT format translator. Then all sub-classes of this abstract SMT format translator can be inserted into the corresponding SMT back end.

qualifiers. Fig. 4.4 shows the class diagram according to the improved design.

Noticed in the new design, Back End only indicate a concrete type of Format Translator that will perform both decoding/encoding tasks for this Back End. Back End only focus on adapting the concrete solver for launching the solving process, and Format Translator will responsible for the details of how to encode type constraints and decode solution from solver. With this improvement, Solver Framework designers can develop multiple SMT Format Translators for different background theories, and plug them into the same SMT Back End that handles the common logic of adapting a concrete SMT solver.

4.3 Z3 Back End with Bit Vector Theory Encoding

Z3 is a state-of-the-art SMT solver developed by Microsoft Research. Z3 integrates a collections of theory solvers, and support many common background theories such as:

Arithmetic Theory (both integer and real numbers), Bit Vector theory, Modeling with Quantifiers, etc. This section introduces the Z3 Back End that integrated Z3 solver into the Type Constraint Solver. Specifically, this section will focus on introducing the implementation on SMT encoding with Bit Vector theory.

The rest of this section is organized as follows: Sec. 4.3.1 gives a brief introduction on power-set-like type qualifier hierarchy, which is the target qualifier hierarchy aimed to solved by Bit Vector theory. Sec. 4.3.2 shows the specification on how to translate between a type qualifier in power-set-like hierarchy and a bit vector. Finally, Sec. 4.3.3 shows the detail encoding on each kind of type constraint for a type system with a power-set-like type qualifier hierarchy.

4.3.1 Power-set-like type qualifier hierarchy

In mathematics, the power set of any set S, is the set of all subsets of S, including the empty set and S itself. A power-set-like type qualifier hierarchy can be defined similarly as follows:

- It has a universe set S that defines all basic atomic elements.
- Each qualifier q in the hierarchy represents a subset of S.
- If q_1 is comparable with q_2 (i.e. they have subtype relationships), then the represented subset s_1 and s_2 is comparable in the context of set operations (i.e. they have subset relationships).

The basic atomic element in the universe set S represents a concept that this type system aims to formalize (e.g. type names in Dataflow type system, and ontic concepts in Ontology type system). As each qualifier represents a set of basic atomic elements, the complete type hierarchy is looked like a power set of the universe set S. Theoretically, the universe set S may have infinite number of basic atomic elements. When running on a program instance, the type system would always just utilize a sub qualifier hierarchy of the theocratic qualifier hierarchy.

4.3.2 Translation between Qualifiers and Bit Vectors

Bit vectors are the basic terms in the encoded SMT formula. To solve type constraints with bit vector encoding, type qualifiers need to be represented as bit vectors. In power-set-like type hierarchy, each qualifier represents a set of basic elements in the universe set. To represent these qualifiers in bit vector form, the encoded bit vectors have to reflect the set of elements that these qualifiers represent. Therefore, a bit vector represents a set of basic elements. Each bit in the vector represents a distinct basic element in the universe set. A bit with value 1 means the corresponding element is belongs to the set represented by this bit vector, and vice versa. In order to precisely indicate the existence of every elements in a bit vector , the length of each bit vector is equal to the size of the universe set. Notice that it is not necessary to pre-encoded **every** qualifiers in the type hierarchy. The only thing need to pre-encoded is the unit bit vectors, that each one represents only one distinct basic element in the universe set. Bit vectors of all qualifiers in the type hierarchy can be computed by the bit operations (*and*, *or*) on these unit bit vectors. For example, given a power-set-like hierarchy with three basic elements $\{A, B, C\}$. One possible encoding for unit bit vectors is to let the left-most bit represents element A, the middle bit represents element B, and the right-most bit represents C:

```
BV_A = 100
BV_B = 010
BV_c = 001
```

With above unit bit vectors, all other qualifiers in the hierarchy can be easily computed. For example, a qualifier represents set $\{A, B\}$ can be computed by the result of disjunction between BV_A and BV_B, i.e. $BV_{Q1} = BV_A \cup BV_B$. As another example, the qualifier represents the empty set is always the zero bit vector, and this also can be computed by the result of conjunctions among all unit bit vectors, i.e. $BV_\emptyset = (BV_A \cap BV_B \cap BV_C)$.

In implementation level, as type system with power-set-like type qualifier hierarchy usually only declare a single type qualifier annotation class, and define member values in the annotation to represent different qualifiers in the hierarchy. This makes implementing a general encoding for translating qualifiers between bit vectors very hard on the Back End level. As a result of trade off, concrete type system is required to provide a translation between qualifiers and bit vectors by implementing the $Z3BitVectorCodec$ interface, as shown in Fig. 4.5.

There are three methods need to be implemented by concrete type systems. First, type system need to tell the Back End the size of the universe set for the given program (as even the theoretical universe set can be infinite large, only a subset of elements will appear in a given program in practice). And then, concrete type system need to implement the two methods of encoding a type qualifier (AnnotationMirror) to the bit vector value represented

```
public interface Z3BitVectorCodec {

    /**
    * Get the fixed Bit Vector size.
    *
    * @return the fixed Bit Vector size
    */
    int getFixedBitVectorSize();

    /**
     * Encode a given AnnotationMirror to a numeric value whose
     * binary representation is the encoded bit vector.
     *
     * @param am a given AnnotationMirror.
     * @return numeral value of the encoded bit vector
     */
    BigInteger encodeConstantAM(AnnotationMirror am);

    /**
     * Decode a given BigInteger value to the corresponding
     * AnnotationMirror.
     *
     * @param numeralValue a BigInteger that represents
     *        a bit vector
     * @return the decoded AnnotationMirror
     */
    AnnotationMirror decodeNumeralValue(BigInteger numeralValue);
}
```

Figure 4.5: The code of $Z3BitVectorCodec$ interface.

by a `BigInteger`, and of decoding the bit vector value represented by a `BigInteger` back to a type qualifier.

4.3.3 Type Constraints Encoding

In SMT Bit Vector encoding, each constraint variable will be encoded as a bit vector variable, and constant variable, i.e. a variable with a constant value to a known qualifier in the hierarchy, will be encoded by concrete type system as a bit vector constant. Type constraints then will be encoded as a SMT formula with boolean predicates on these bit vectors.

Encoding for Subtype Constraint

For a type system with a power-set-like type qualifier hierarchy, a subtype constraint represents a subset relationship between the basic elements that the constraint variables represented. The direction of subset relationship is determined by the concrete type qualifier hierarchy, i.e. whether the super type represents a super set of the set represented by the subtype, or in a reversed direction. Therefore, a subtype constraint is encoded as a subset constraint on the two bit vector variables corresponding to the constraint variables in the subtype constraint. The direction of the encoded subset constraint is determined by the concrete type qualifier hierarchy. A subset constraint between two bit vectors are expressed by union and intersection operations. Specifically, given two bit vectors B_1 and B_2, we can compute their unions $B_u = B_1 \cup B_2$ and intersection $B_I = B_1 \cap B_2$. if B_1 and B_2 has subset relationship, then the one represents the subset should equal to their intersection set, and the one represents the superset should equal to their union set. Furthermore, an assertion on only one of these two cases should be enough, as they have to be both satisfied or both failed.

Therefore, the algorithm of encoding a subtype constraint would be:

Algorithm 1 Algorithm for encoding subtype constraint

```
1: procedure EncodeSubtypeConstraint(subtypeBitVector, supertypeBitVector)
2:     $superset = subtypeBitVector \cup supertypeBitVector$
3:     $assert(predicate(superset\ equal\ to\ subtypeBitVector))$
```

The boolean flag *supertypeIsSuperSet* indicates the direction of the type hierarchy. The default value of *supertypeIsSuperSet* is true, and concrete type system can override the value to indicates the direction of its type hierarchy.

Algorithm 2 Algorithm for encoding equality constraint

```
1: procedure EncodeEqualityConstraint(bitVector1, bitVector2)
2:     assert(predicate(bitVector1 equal to bitVector2))
```

Encoding for Equality and Inequality Constraint

As constraint variables are directly encoded as bit vectors, the equality and inequality type constraints between constraint variables can be directly encoded as boolean predicates between the bit vectors. I.e. the equality type constraint between variable v_1 and v_2 will be directly encoded as the equality boolean predicate between bv_1 and bv_2. Similarly, Inequality type constraint will be encoded as the inequality boolean predicate between the corresponding bit vectors.

Encoding for Comparable Constraint

A Comparable constraint means the two constraint variables have subtype relationships, but do not specify which are the super type and which are the subtype. Therefore, given a comparable constraint between variables V_1 and V_2, the encoding is a disjunction of the case "V_1 is subtype of V_2" and "V_1 is super type of V_2". The detail encoding algorithm is:

Algorithm 3 Algorithm for encoding comparable constraint

```
1: procedure EncodeComparableConstraint(firstBitVector, secondBitVector)
2:     $subset = firstBitVector \cap secondBitVector$
3:     $superset = firstBitVector \cup secondBitVector$
4:     $firstIsSuper$ = predicate(superset equal to firstBitVector)
5:     $firstIsSub$ = predicate(subset equal to firstBitVector)
6:     assert($firstIsSuper \vee firstIsSub$)
```

Encoding for Preference Constraint

The preference constraint in the Checker Framework Inference is always between a variable and a constant. The meaning of preference constraint is to express the preference on preferring variable v to have a solution exactly as constant c. Therefore, the preference constraint is treated as a soft equality constraint with a weight. The encoding of preference constraint is similar to the equality constraint, just with an additional weight to expressing the extend of preference. As preference constraint is a soft constraint, Z3 will try to find a solution that satisfies all other constraints and also maximized the number of these soft constraints to be satisfied, instead of try to always find a solution satisfies all the constraints.

4.4 Conclusion

This chapter discussed the improvement on *Type Constraint Solver* [11]. We extended the *Type Constraint Solver* from only supporting SAT solvers to also supporting SMT solvers. A new SMT back-end based on *Z3* solver has been implemented and integrated into the *Type Constraint Solver*. Also, a new SMT encoding with Bit Vector Theory is proposed for type systems with a power-set-like type hierarchy.

Chapter 5

Improvements to the Dataflow Type System

This chapter discusses the improvement of applying the SMT encoding with Bit Vector theory introduced in Chap. 4 to the Dataflow type system (abbreviated as Dataflow in the rest of this chapter). Dataflow is previous work [11] that reasons about concrete run-time types of components in programs.

The rest of this chapter is organized as follows: Sec. 5.1 gives a brief introduction to Dataflow. It discusses the type hierarchy, and introduces the solving approach in previous work. Sec. 5.2 motives the improvement of applying the SMT encoding to Dataflow, and discusses the Bit Vector encoding for Dataflow qualifiers in detail. Sec. 5.3 gives the experimental results of comparing the new SMT approach with the previous approach. Sec. 5.4 discusses some possible future works for Dataflow. Finally, Sec. 5.5 concludes.

5.1 Background on Dataflow Type System

This section introduces a brief background of Dataflow that is necessary for understanding the new SMT solving approach based on Bit Vector Encoding for Dataflow. A more detailed introduction on Dataflow type system can be found in the original Master's thesis [11]. In the rest of this section, Sec. 5.1.1 introduces Datatlow qualifiers and the type hierarchy. Sec. 5.1.2 introduces Dataflow annotated base cases, which are the ground truth for type inference of Dataflow used in both graph-solving approach and the new SMT approach. Sec. 5.1.3 describes previous constraint graph based solving approach.

5.1.1 Dataflow Qualifiers and Qualifier Hierarchy

As Java supports type polymorphism, the run-time types of fields, methods, parameters, and local variables may vary from their declared types. Dataflow reasons about the concrete run-time types for type declarations in programs. Fig. 5.1 gives an example for illustrating the purpose of Dataflow.

```java
Object foo(boolean b) {
  if (b) {
        return "aString";
  }
  return Integer.valueOf(1);
}
```

Figure 5.1: An example illustrating the purpose of Dataflow.

The return type of method `foo` is declared as `Object`. Dataflow reasons about the concrete run-time return types are either `String` or `Integer`. To represent the run-time types anaylsis result for declare types, Dataflow uses a single Java Annotation `@Dataflow` with two parameters:

- *typeNames*: indicates an over-approximation of all possible run-time types of the annotated object.
- *typeRoots*: indicates an over-approximation of all possible upper bounds of run-time types of the annotated object.

For the code example in Fig. 5.1, Dataflow will infer `@Dataflow(typeNames={"String", "Integer"})` on the return type of method `foo`.

Notice that an inferred `Dataflow` qualifier may have overlaps between its `typeRoots` and `typeNames`. In this case, the more general one will be kept, i.e. an upper bound in `typeRoots` subsumes all subtypes in `typeNames`. For example, an inferred Dataflow qualifier with `typeRoots` "Object" and `typeNames` "String" will be simplified as a Dataflow qualifier with only `typeRoots` "Object", as upper bound "Object" subsumes the concrete run-time type "String".

The subtype relationship between Dataflow type τ_1 and τ_2 is: τ_1 is a subtype of τ_2, if and only if values in `typeNames` and `typeRoots` of τ_1 can be bounded by the values in

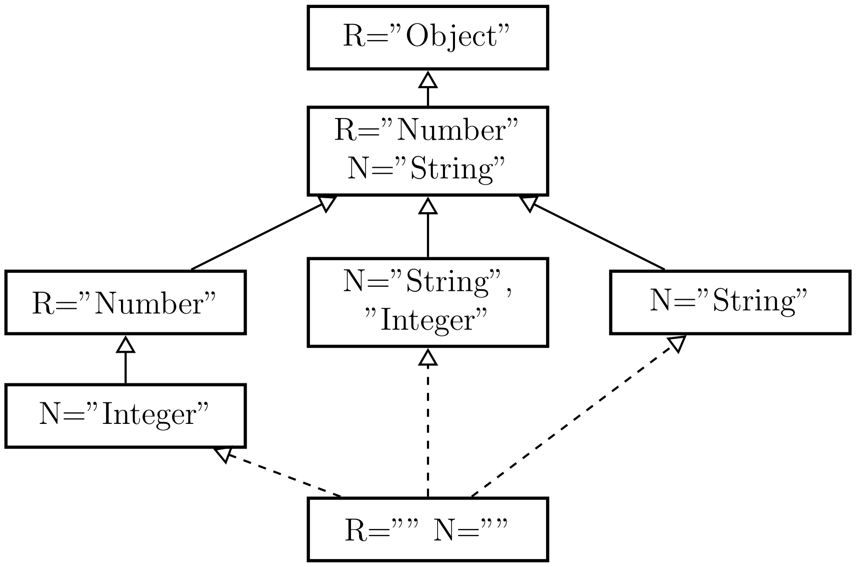


Figure 5.2: A partial qualifier hierarchy of Dataflow type system for value set of `typeRoots`={Object,Number} and `typeNames`={Integer,String}. In each node, R means `typeRoots`, and N means `typeNames`. Solid arrow represents a strict subtype relationship between two nodes in this partial qualifier hierarchy, and dashed arrow represents a non-strict subtype relationship.

τ_2. For example, `@Dataflow(typeRoots="Object")` is the top qualifier in the hierarchy, as values in any other Dataflow qualifier are bounded by `Object`.

Theoretically, the type hierarchy of Dataflow is infinite, as there are infinite concrete run-time types and upper bounds may appear in programs. However, the shape of Dataflow type hierarchy is always power-set-like; i.e. given an universe set consists of all possible run-time types and upper bounds in a program, any subset of the universe set can be represented by a distinct Dataflow qualifier in the type hierarchy. As an example, Fig. 5.2 shows the sub-hierarchy on values set: `typeRoots`={"Object", "Number"} and `typeNames`={"String", "Integer"}.

5.1.2 Base Cases

Base cases are those Java expressions that can directly determine their concrete run-time types in programs. These expressions are `Literals`, `Class Instance Creations`, and `Array Creations`. In addition, although `Method Invocation` for methods from byte code cannot provide information of the concrete run-time types they return, these expressions can provide an upper bound of the run-time return types, therefore `Method Invocation` for byte code methods also counts as a base case.

Dataflow reasons about run-time types of other places in programs by propagating these base cases. The base case propagation is done by solving the type constraints generated according to the type rules.

5.1.3 Constraint Graph Based Solving Approach

As Dataflow type system has infinite qualifiers in theory, and usually uses a lot of qualifiers (>10, sometimes even >100) in practice for a given program, the general SAT encoding approach [11] cannot solve type constraints for Dataflow type system in a reasonable time. To resolve this time-out problem, a constraint graph based solving approach is proposed [11].

The basic idea of graph-solving approach is to compute the reachability of each base case to the variables in their super type chain. A super type chain of a base case consisted of all variables that are transitively constrained as super types of this base case by subtype constraints. If a variable is on the super type chain of a base case, then the run-time type represented by the base case could ”flow” into this variable. Therefore, the solution of a variable is the union set of the base cases that can reach this variable through a super type chain. Fig. 4 shows an algorithm for illustrating the graph-solving approach. Notice that in [11] the graph-solving approach is designed as a separate-and-merge algorithm. However, the graph-solving algorithm described here is essentially the same as the one in [11], and this one is easier to illustrate and understand than the separate-and-merge algorithm described in [11].

It is worth to notice that the graph solving approach will ignore some variables in a constraint graph. Specifically, variables that are not in any super type chain of any base case will be ignored. Fig. 5.3 shows an example constraint graph for illustrating which variables can be solved and which variables are ignored by graph-solving approach.

Ignoring variables that are not in any super type chain of base cases actually weakens the meaning of Dataflow qualifier — a Dataflow qualifier only reflect the run-time types that Dataflow can observe, therefore it reflects a partial-approximation instead of an over-approximation of the annotated object. Fig. 5.4 shows an example for illustrating this limitation of graph-solving approach. In this example, string literal `"str"` and method parameter `boo` are both assigned to local variable `bar`. Therefore, all possible run-time types of variable `bar` should be the union set of `String` type and all possible run-time types

Algorithm 4 Algorithm of Graph-solving Approach

```
1: procedure GraphSolving($variables, baseCases$)
2:     Initialize $solutions$ as an empty map
3:     for each $variable$ in $variables$ do
4:         $solution = \emptyset$
5:         for each $runtimeType$ in $baseCases$ do
6:             if isReachableBySuperTypeChain($runtimeType$, $variable$) then
7:                 $solution \leftarrow solution \cup runtimeType$
8:         if $solution \neq \emptyset$ then
9:             $solutions \leftarrow variable, solution$
    return $solutions$
```

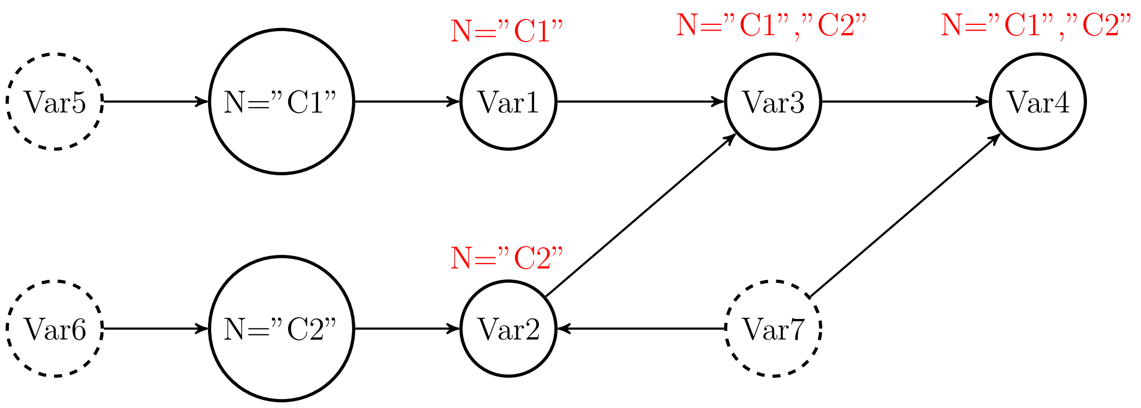
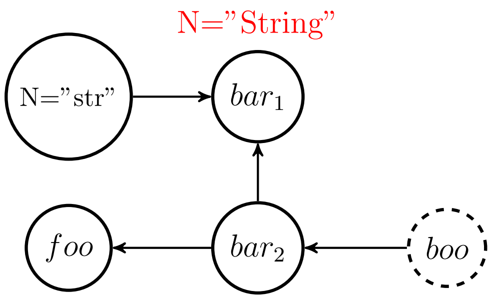


Figure 5.3: An example graph for illustrating the graph-solving approach. Each arrow indicates a subtype relationship from subtype point to super type. The label of each variable node with red text is the solution of that variable given by graph-solving approach. Variable nodes in dashed shape are the variables ignored by graph-solving approach.

```
1 /*foo*/ Object foo(Number boo) {
2   /*bar1*/ Object bar = "str";
3   /*bar2*/ bar = /*boo*/ boo;
4   return bar;
5 }
```

(a) Code example.



(b) The constraint graph.

Figure 5.4: An example for illustrating why the graph-solving approach weakens the meaning of Dataflow qualifiers. Node `bar1` is the constraint variable on `bar`'s declaration, and node `bar2` is a refined constraint variable represents the data-flow refinement result of the new assignment context in line 3. Notice the solution of local variable `bar` given by graph-solving approach is `typeNames`="String", and node `boo` is ignored.

of parameter `bar`. If method `foo` never get called in the analyzed program, the solution of `bar` given by graph-solving approach would be `typeNames` "String", and the effect of `boo` on `bar` is ignored. Therefore, the solution of `bar` given by graph-solving approach is a partial-approximation if no invocation of method `foo` is observed by Dataflow. Similarly, for programs that partially annotated with Dataflow qualifiers, graph-solving approach cannot enforce the variables on the subtype chains of these manually annotated base cases. Consequently, graph-solving approach does not support annotated programs.

5.2 The Improvement: SMT Encoding Approach for Dataflow

5.2.1 Bit Vector Encoding for Dataflow qualifiers

As discussed in Chap. 4, using Bit Vector encoding for a type system with a power-set-like qualifier hierarchy can reduce the solving time of type constraints into a reasonable range. Dataflow qualifiers follow a power-set-like qualifier hierarchy, and subtype relationships between Dataflow qualifiers can be translated to subset relationships between the elements (run-time types) represented by these qualifiers. Therefore, to resolve the limitation of graph-solving approach (ignoring variables that are not on super type chains of base cases), the bit vector Encoding introduced in Chap. 4 has been applied to Dataflow type system.

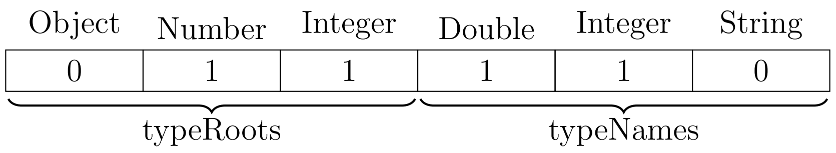


Figure 5.5: A example of Bit Vector encoding of @Dataflow(typeRoots="Number") under the universe set: `typeRoots`={"Object","Number", "Integer"} `typeNames`={"Double", "Integer", "String"}. Notice that type `Integer` takes two bits in the Bit Vector, one represents `Integer` as a run-time type upper bound (it is possible if a byte code method returns an integer), and one represents `Integer` as a concrete run-time type.

The bit vector encoding for Dataflow qualifiers follows the encoding specification introduced earlier in Sec. 4.3.2. For a given program, the bit vector size is the sum of the number of `typeRoots` and the number of `typeNames` that appear in this program. For each Dataflow qualifier that represents a single `typeName`, a unit bit vector is encoded to represent this `typeName`. For each Datflow qualifier represents a single `typeRoot`, the encoded bit vector represents all run-time types in the universe set that are bounded by this `typeRoot`. Therefore, the encoded bit vector is the union result of the unit bit vector represents this `typeRoot`, and all unit bit vectors that represents a `typeName` or a `typeRoot` that is a subtype of this `typeRoot`. Fig. 5.5 shows an example of encoding of a Dataflow qualifier represents a single `typeRoot`.

The type constraints encoding is the same as described in Sec. 4.3.3.

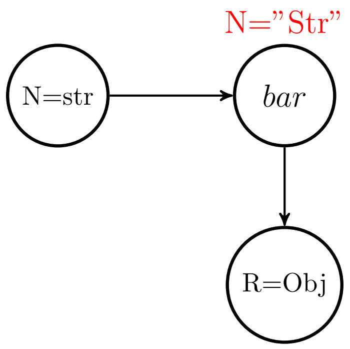
5.2.2 Preference Tuning on Subtype Constraints

Dataflow type system is under-constrained, i.e. for a given program, the type constraints are not expressive enough to guide the solver to give a desired solution. For example, given a subtype constraint in which constraint variable `var` is subtype, constant `typeRoot` "Number" is super type. Suppose `var` was not on any super type chains of base cases. An empty solution (bottom) is valid for `var`, but it reflects no useful information. In contrast, propagates the type bound `typeRoot` "Number" to `var` as its solution would provide more information of `var`.

In order to guide solver to give a desired solution, subtype constraints are tuned by preferences. To propagate base cases as much as possible, subtype variables are preferred to have the same solution as their super type variables. By preferring equal solutions on variables in subtype constraints, variables that connects to base cases by subtype con-

```
void foo() {
  /*bar*/ Object bar = "str";
}
```

(a) Code example.



(b) The constraint graph.

Figure 5.6: An example that explains why case *constant* <: *variable* has more weight of equality preference than the case *variable* <: *constant*.

straints are preferred to have a solution that equals to the base case they are connected to.

Specifically, different weights are made for different compositions of variables for a subtype constraint:

- case **constant** <: **variable**: Prefer *variable* equals to *constant* with weight 3.
- case **variable** <: **constant**: Prefer *variable* equals to *constant* with weight 2.
- for all *variable* in subtype constraints: Prefer *variable* equals to BOTTOM with weight 1.

Notice the case of *constant* <: *variable* has more weight of this equality preference than the case of *variable* <: *constant*. This is because for the case of a variable directly connects to both a super type constant, and a subtype constant, the subtype constant is preferred to be the solution of this variable as it contains more specific information. Fig. 5.6 shows an example for illustration.

Preference all *variables* to BOTTOM is for avoiding the solver gives random solutions on orphan variables (variables that do not connect to any base cases).

5.2.3 Type Declaration Bound

As bit vector approach is able to propagate type bounds in base cases to constraint variables in their subtype chains, it is reasonable to constrain each constraint variable with its

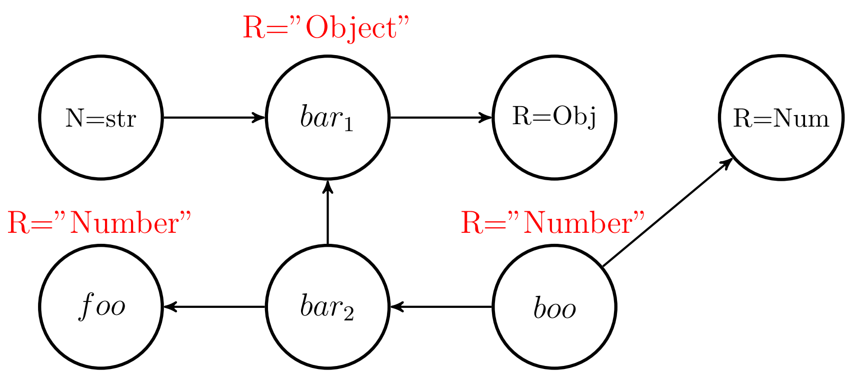


Figure 5.7: The new constraint graph with type declaration bound for code example in Fig. 5.4. The label above each variable node with red text is the solution given by bit vector encoding approach.

declaration type in programs. This has two benefits: 1) avoid solver gives a solution on a variable that conflicts with its declaration type. For example, inferring a solution with run-time type `Number` to a variable of `String` type. 2) If Dataflow observes no concrete run-time types for a given variable, at least its type declaration can be provided as an upper bound of its run-time types. Fig. 5.7 shows the constraint graph of code example with the newly added type declaration bound constraints in Fig. 5.4. Fig. 5.8 shows a comparison with inferred result by graph-solving approach and by bit vector approach with declaration bound constraints for the previous code example.

Notice that the bit vector approach doesn't give the best solution for local variable `bar` due to the preference tuning on preferring `bar` equal to its type declaration bound. This problem can be solved by adding a more desired preference to prefer each constraint variable in the super type chains of base cases has a solution with the smallest set of run-time types (i.e. prefer variables equal to the least upper bound of these base cases). However, how to encode this preference is still need further research.

5.3 Experiment

We evaluate Dataflow in 4 un-annotated scientific libraries. Fig. 5.9 shows the size of each library in the benchmark. The number of Java files, comments, blanks, and lines of source code are computed by the `cloc` tool. To give a more straightforward observation of the relation between project size and the sizes of constraints and variables generated for each project, the size of constraints and variable slots are also shown in the table.

```java
Object foo(Number boo) {
  @DataFlow(typeNames={"String"})
  Object bar = "str";
  bar = boo;
  return bar;
}
```

(a) Inferred result of graph-solving approach

```java
@DataFlow(typeRoots={"Number"}) Object foo(
  @DataFlow(typeRoots={"Number"}) Number boo) {
  @DataFlow(typeRoots={"Object"}) Object bar = "str";
  bar = boo;
  return bar;
}
```

(b) Inferred result of bit vector encoding approach with type declaration bound

Figure 5.8: The inferred results of graph-solving approach and bit vector approach on the code example in Fig. 5.3.

Benchmark	Project Size				Constraint and Slot sizes		
	Files	Blank	Comment	Java LoC	Variable Slots	Constraints	
						Subtype	Equality
imagej	14	220	694	796	82	47	6
FaceDetection	27	320	141	1074	549	717	88
jama	17	1055	1817	4599	2974	4159	550
jblas	117	6096	12215	22378	13787	22469	1349

Figure 5.9: Size of projects, and corresponding generated constraint and slot sizes for the Dataflow Type system.

Benchmark	Slot	Constraint	Encoding Time (ms)			Solving Time (ms)		
			Seq.	Par.	B.V.	Seq.	Par.	B.V.
imagej	82	53	84	2	15	6546	647	57
FaceDetection	549	811	2	5	58	30212	1111	582
jama	2974	4712	9	119	74	15093	721	2532
jblas	13787	23853	14	373	246	63408	2662	357939

Figure 5.10: Timing result of Running Dataflow with different settings. Text in olive color represents the result of using Graph-soling approach with sequentially executing solvers (Seq.). Text in blue color represents the result of using Graph-soling approach with executing solvers in parallel (Par.). Text in red color represents the result of using the bit vector approach with Z3 back-end (B.V.). Run on Mac OS, with 2.3 GHZ Intel Core i7 processor, 4 cores, 16 GB memory.

To evaluate the timing influence of the bit vector approach, we run Dataflow with three different settings: 1) using graph-solving approach with sequentially executing SAT solvers, 2) using graph-solving approach with executing SAT solvers in parallel, and 3) using bit vector encoding approach with z3 solver. Fig. 5.10 shows the timing result. Specifically, the timing result of running SAT solvers sequentially is the accumulative result of all solvers' encoding time and solving time. For the timing result of running SAT solvers in parallel, the solving time is computed as the time difference between the first solver has been started, till the last solver has finished its solving. The encoding time is computed as the maximum encoding time among these solvers.

The timing result suggests that bit vector approach needs more time for encoding, and has the best solving time performance on the two small projects (<5k LoC). However, for the two larger projects, Bit Vector approach adds non-trivial solving time overhead compared to the graph solving approach with parallel setting. In detail, for the two small projects, the Bit Vector approach faster 83 times than the graph solving approach with sequential setting, and faster 6.6 times than the parallel setting. However, for the two larger projects, on average the Bit Vector approach add 68.9 times overhead, and add 134.4 times overhead for the largest project `jblas` than the graph solving approach with parallel setting.

Fig. 5.11 shows the inference result of running Dataflow by using graph-solving approach and by using bit vector encoding approach respectively. To give a more concrete comparison of the inference result, we filter out three kinds of solutions that are not interested: 1) TOP solutions, 2) BOTTOM solutions, and 3) Primitive solutions. TOP and BOTTOM are filtered out as they do not provide useful informations. Primitive solutions are filtered out because primitive types do not support polymorphism, hence primitive solutions are not interesting. For every project, we manually examine the inference results of two approaches, and verified that both approaches give a correct and meaningful result.

Generally, since the Bit Vector encoding approach does not ignore any variables, it propagates more Dataflow annotations in projects. On average, the bit vector encoding approach propagates 1.56 times more interesting Dataflow annotations than the graph-solving approach. In detail, through the manually examination on the inference results, the Bit Vector encoding approach is able to give `typeRoots` information on the return type of methods that Dataflow does not observe any invocations of these methods, while the graph solving approach does not provide any information. This result suggests the bit vector encoding approach is able to reason about more run-time types information than the graph-solving approach.

5.4 Future Work

There are several possible future work to further improve the Dataflow type system.

First, the current preference tuning is actually an "equality" preference. The equality preference would become ineffective for those variables on the intersections of multiple base cases, as none of the preference of these variables to be equal to a single base case can be satisfied. For these variables, a more desired preference tuning would be a "least upper bound" preference, so that the most specific solution is preferred

Also, the current preference tuning does not takes advantage of the information in constraint graph. For example, instead of propagating type bounds to all variables on their subtype chains, we only propagate type bounds to the variables that are not on any super type chain of base cases. This can guide solver to give more precise solutions on the variables that are on the super type chains of base cases.

Finally, during our development, the type declaration bound implementation keep running into corner cases for complicated projects. For the 15 scientific libraries, the Bit Vector approach with the type declaration bound crashed on 11 projects. We will try to fix the bugs behind these crashes, and make the system robust.

Benchmark	LoC	Variable Slot	Total Inferred annotations		Interested		Non Interested		Types		TypeNames		TypeRoots	
			G.	B.V.	G.	B.V.	G.	B.V.	G.	B.V.	G.	B.V.	G.	B.V.
imagej	796	82	22	38	22	37	0	1	6	11	5	5	2	8
FaceDetection	1074	549	347	439	147	284	200	155	32	42	23	26	15	26
jama	4599	2974	2172	2557	237	1261	1935	1296	16	24	14	19	5	9
jblas	22378	13787	8140	12305	3376	4450	4764	7855	77	79	74	74	17	22

Figure 5.11: Inference result for the Dataflow type system. Text in blue color represents the result of using graph-solving approach (G.), text in red color represents the result of using bit vector encoding approach (B.V.). `Non Interested` is the number of non interested solutions that are filtered out (TOP, BOTTOM, and Primitives). `Types` are the total types (union set of `TypeNames` and `TypeRoots`) that Dataflow reasons about in these projects.

5.5 Conclusion

This chapter discusses the improvements for the Dataflow type system. We discuss the limitation of previous graph-solving approach in detail, and show how the bit vector encoding approach can improve the Dataflow inference results compared to the graph-solving approach.

For experimentation, we run Dataflow on 4 real world projects. The result suggests the bit vector encoding approach adds non-trivial time overhead for solving large projects (> 5K LoC) compared to the graph-solving approach with parallel execution setting. However, the bit vector encoding approach is able to propagate more run-time type information than the graph-solving approach.

Chapter 6

Ontology Type System

This chapter introduces the Ontology type system. Ontology type system (abbreviated as Ontology in the rest of this chapter) and its inference is used to reason about a coarse abstraction for a given program based on ontic concepts. One straightforward example for illustrating ontic concept is the SEQUENCE concept. SEQUENCE represents a collection of elements organized in some orders. Ontology groups Java Array type, List, and List subtypes to SEQUENCE concept. With Ontology inference, a program abstraction with ontic concepts is produced, and concrete Java types are grouped by ontic concepts. The goal of Ontology is to produce reasonable semantic abstractions for programs, so that the produced abstractions may facilitate other program analysis tools on their tasks.

The rest of this chapter is organized as follows: Sec. 6.1 motivates Ontology, and gives a basic overview on the type system. Sec. 6.2 describes Ontology qualifiers and the type hierarchy in detail. Sec. 6.3 discusses the type inference for Ontology, which is a key process to produce program abstractions with ontic concepts. Sec. 6.4 summarizes the implementation of Ontology, and discusses on how to integrate Ontology with other program analysis tools. Sec. 6.5 presents the experimental result of Ontology on 15 scientific libraries. Sec. 6.6 discusses some possible future work for Ontology. Finally, Sec. 6.7 concludes.

6.1 Introduction

With the increasing amount of available source code on the Internet, many approaches [19, 14, 22, 12, 17, 10] have been proposed to retrieve, reuse, and repair code based on the understanding of given programs. Building proper abstractions that remain important/interesting properties and meanwhile striped out trivial noises from concrete programs is a key foundation that many of these approaches rely on. Types, as a natural and powerful approach to classify the objects that a program manipulates, contains valuable informations that may reflect important properties of programs. However, efficiently utilizing types for enhancing comprehension among multiple programs is difficult. Different programs/libraries may use different types to achieve similar logic, organize similar objects, and represent similar components. Therefore, many efforts have been dedicated on researching how to utilize types for mining useful informations from programs [6, 15, 16].

This chapter presents the Ontology type system, that proposes a new way of categorizing types by ontic concepts. For a programming language, Ontology group types in two dimensions. First, it groups similar types to more general concepts (ontic concepts) that reflect essential semantic properties of these types. For an illustration of grouping similar types into ontic concepts in Java, *Array* type and *List* type are similar. Ontology groups both *Array* type, *List* type, and subtypes of *List* to SEQUENCE ontic concept. In another dimension, Ontology propagates domain-specific concepts mined by other program analysis tools. For a given program, a set of ground truths that labels fields with domain-specific concepts is obtained from other program analysis tools. Then Ontology propagates this ground truth set on variables, functions, and interfaces in the program. For a simple illustration, given a physic library, suppose Ontology obtains a ground truth of FORCE concept on a field f of type $Vector$, i.e. this field f is considered to relate to FORCE concept. As a result, Ontology propagates the FORCE concept from this field f to other related places in the program by type inference. The produced inference result is guaranteed to be sound according to the type rules and type hierarchy in Ontology. Therefore the result is ensured to reflect program constructs that are related to the propagated domain-specific concepts.

The focus of Ontology type system is to provide an infrastructure of ontic concept propagation. Clients of Ontology can customize on what types should be considered as similar types in their programs, and what are semantic concepts generalized from these types. Also, client of Ontology can specify the domain-specific concepts that are desired to be propagated in programs. Sec. 6.3.1 introduces the two dimensions of ontic concepts in detail, and Sec. 6.4 provides concrete instruction of how to import customized ontic concepts into Ontology.

The produced result of Ontology on a given program is a semantic abstraction that types are annotated with ontology qualifiers that represent specific ontic concepts. Sec. 6.2 introduces the meaning of ontology qualifiers in detail. With real types grouped by ontic concepts, clients of Ontology may be able to facilitate their program analysis tasks by analyzing on higher-level semantic abstractions.

6.2 Type Qualifiers and Qualifier Hierarchy

This section introduces ontology qualifiers and its qualifier hierarchy. Sec. 6.2.1 introduces the meaning of ontology qualifiers and the qualifier hierarchy. Sec. 6.2.2 introduces the polymorphic qualifier for Ontology, which is helpful for generating context-sensitive constraint variables.

6.2.1 Ontology Qualifiers and Qualifier Hierarchy

Ontology encodes ontic concepts as enum values, and uses a single Java annotation class @*Ontology* that takes an enum array of these ontic concepts as an argument. Hence, all normal ontology qualifiers are represented by a single Java annotation @*Ontology*. An ontology qualifier with a group of ontic concepts is an under-approximation of the annotated object. An under-approximation means this approximation reflects some precise information on the approximated object, but may not reflect all information. In contrast, over-approximation gives an upper bound of the information that the approximated object can express, but not sure what the exact information the object is expressed. As the desire of Ontology is to provide precise information of objects in programs, therefore under-approximation is chosen for ontology qualifiers. For example, if an object is annotated with @*Ontology*($\{SEQUENCE, FORCE\}$), it means this object is ensured that it is related to SEQUENCE and FORCE concepts. However, this object may possible also related to other concepts. Therefore, the type hierarchy starts from an empty under-approximation, and given a collection of constraints, the goal is to find the lowest satisfied fixed-point that is expressed the most precise under-approximation. Specifically, the subtype relationship in Ontology type system is: Ontology type τ_1 is subtype of Ontology type τ_2, if the set of ontic concepts represented by τ_1 is a super set of the set represented by τ_2. For example, @*Ontology*($\{SEQUENCE, FORCE\}$) is a subtype of @*Ontology*($\{SEQUENCE\}$).

Theoretically, the type hierarchy of Ontology is infinite, as there are infinite ontic concepts can be expressed. However, the type hierarchy of Ontology always follow a power-set-like shape, and Fig. 6.1 shows a sub-hierarchy on three ontic concepts SEQUENCE (SEQ), VELOCITY (VEL), FORCE (FOR).

@*Ontology*$\{TOP\}$ is the top qualifier in the hierarchy. It represents an empty set of ontic concepts. The BOTTOM qualifier in the type hierarchy is a theoretical lowest

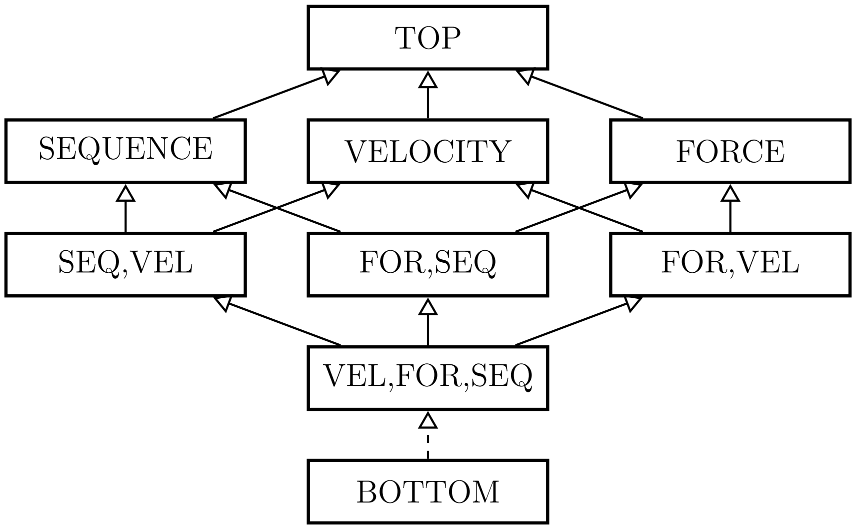


Figure 6.1: A sub-hierarchy of Ontology Type System on three ontic concepts: SEQUENCE (SEQ), VELOCITY (VEL), and FORCE (FOR).

bound to make the type lattice complete. In practice, objects in programs should never be annotated with @*Ontology*($\{BOTTOM\}$).

Since @*Ontology*$\{TOP\}$ represents an empty under-approximation, annotating every location by @*Ontology*$\{TOP\}$ is a valid solution. However, this is an useless solution, as it does not give any useful approximation about the given program. Sec. 6.3.3 describes how subtype constraints in Ontology are tuned by preference to infer a desired solution.

6.2.2 Polymorphic Qualifier: PolyOntology

Usually, there are some classes in a library that encapsulate some common logics needed by other components in this library. The methods in these classes are general and do not related to specific ontic concepts. Fig. 6.2 shows an example of a code snippet from a physic library.

```
class VectorFactory {
  Vector createVector() {
    // Some initializations...
    return new Vector();
  }
}
```

Figure 6.2: A code snippet from a physic library.

The `createVector` method in `Vector` class is a general method that is commonly used by other components in the physic library. Consequently, Ontology may infer unexpected result on this common `createVector` method, if the `createVector` method was used by multiple callers that are related to different ontic concepts, for example:

```
@Ontology({FORCE}) Vector force;
@Ontology ({VELOCITY}) Vector velocity;
//...
force = vectorFactory.createVector();
velocity = vectorFactory.createVector();
```

```
@SEQUENCE List<Integer> swapsort(@SEQUENCE List<Integer> list) {
  int n = list.size();
  int k;
  for (int m = n; m >= 0; m--) {
    for (int i = 0; i < n - 1; i++) {
      k = i + 1;
      if (list.get(i) > list.get(k)) {
        int temp;
        temp = list.get(i);
        list.set(i, list.get(j));
        list.set(j, temp);
      }
    }
  }
  return list;
}
```

```
@VELOCITY Vector externalVelocity;
@FORCE Vector externalForce;

public void applyVelocity(@VELOCITY Vector velocity) {
  externalVelocity.add(velocity);
}

public void applyForce(@FORCE Vector force) {
  externalForce.add(force);
}
```

Suppose the constraint variable on the return type of `createVector` method declaration is v_1. The two assignment statements in above code cause two subtype constraints on v_1: v_1 <: $FORCE$ and v_1 <: $VELOCITY$. Consequently, Ontology inferred v_1 as @*Ontology*({*FORCE*, *VELOCITY*}). This is an unexpected result, as the `createVector` method is a common method, and does not related to any specific ontic concepts. The reason of this unexpected result is because Ontology doesn't consider invocation context when generating type constraints. To resolve this problem, @*PolyOntology* is introduced.

@*PolyOntology* is a polymorphic qualifier that follows the qualifier polymorphism defined in the Checker Framework. Polymorphic qualifier allows a single method to have multiple different type signatures, depending on the invocation context. For example, with the `createVector` method annotated by @*PolyOntology*, for each invocation of `createVector` method, a new constraint variable will be created for the return type of `createVector` method. The new constraint variable represents the invocation context, and decouples the invocation context with the method declaration. Consequently, the two subtype constraints for above example would be: v_2 <: $FORCE$ and $v_3 < VELOCITY$, where v_2 represents the first invocation context, and v_3 represents the second invocation context.

Currently, Ontology cannot infer @*PolyOntology* automatically. Clients of Ontology need to annotate polymorphic methods with @*PolyOntology*, to represents the return type and method parameters in this method are polymorphic, and do not related to concrete ontic concepts. A future work would be designing constraints to enable Ontology infer @*PolyOntology* for programs.

6.3 Type Inference for Ontology

This section introduces the type inference approach for Ontology. Type inference is a key process that propagates the customized ontic concepts from clients of Ontology. The result of this concept propagation by type inference is a semantic abstraction of a given program, in which real types are grouped by ontic concepts.

The rest of this section is organized as follows: Sec. 6.3.1 discusses the two dimensions of customized ontic concepts that clients of Ontology can customized on. Then Sec. 6.3.2 introduces what kinds of type constraints are generated in Ontology. Finally, Sec. 6.3.3 describes the Bit Vector solving approach for solving the type constraints.

6.3.1 Annotated Base Cases

The type inference of Ontology is actually a process of propagating ontic concepts that come from base cases. Base cases are ground truths as pre-knowledges for Ontology that mapping some places in programs to ontic concepts. Base cases come from two dimensions:

Type Summarization Rules

A type summarization rule defines a mapping from a group of real types to a corresponding ontic concept. For a given real type τ, if a type summarization rule summarizes this real type to an ontic concept C, then Ontology will annotate `@Ontology({C})` on all variables, creation expression whose type is a subtype of τ.

Ontology has two built-in type summarization rules. The first one summarizes Java *Array* type, *List*, and subtype of *List* to SEQUENCE concept. The second one summarizes *Dictionary*, *Map*, and their corresponding subtypes to DICTIONARY concept.

In addition, clients of Ontology can insert additional type summarization rules. One use scenario of inserting additional type summarization rules is for generalized type search in multiple libraries. For type search engines, one common use case is users desire to learn the usage of functions in libraries that operating on some specific concepts. The problem is users cannot perform this type search without the knowledge of the concrete type that represents the corresponding concept used in the library. Even worse, different libraries may use different types to represent the same theoretical concept. For example, an user want to learn how mathematical libraries operating on Matrix. However, library A uses a customized type `MatrixA` to represent Matrix in their library, while library B use type

`MatrixB`. A type summarization rule summarizes both `MatrixA` and `MatrixB` to a common ontic concept MATRIX would be helpful to enable users perform type search in these libraries on the same high level concepts.

Fields Annotated by Domain-specific concepts

Usually programmers assign meaningful name to variables, method, and fields in order to make their program more understandable. Clients of Ontology may utilize this fact, and pre-annotate fields with ontic concepts that reflect the interesting meaning of these fields. For example, given a physical library, if a field named "externalForce", then most likely this field is related to FORCE concept, and the usage of this field is related to some logic operating on FORCE. If clients of Ontology can provide a set of ground truths on fields in the program, where these fields are annotated by proper ontic concepts, then Ontology will utilize these ground truths, and propagate ontic concepts on the ground truths to other related places in the program.

The reason of requiring ontology annotations on fields instead of on other places is because Ontology only need a minimum set of ground truths to propagate ontic concepts. Therefore, instead of requiring clients of Ontology analyze every places in the program, only field analysis is required for obtaining the ground truths of these domain-specific concepts.

6.3.2 Constraint Generation

For a given program, the Checker Framework Inference generates a set of type constraints according to type rules defined in the type system. Ontology uses the standard type rules in the Checker Framework, and does not add additional type rules. Therefore, only subtype constraints and equality constraints are generated in Ontology on a given program. In these constraints, the constraint variables for base cases are created as constants with the corresponding ontic concepts as values. Therefore, the generated set of constraints are type constraints between constants of base cases and variables of other places in the program.

6.3.3 Bit Vector Based Solving Approach

Once type constraints are generated, these constraints are encoded as a SMT problem with Bit Vector theory. Then Z3 BackEnd try to find a solution for the given SMT problem.

Finally, if a solution is found, it is translated back from bit vectors to ontology qualifiers that will be inserted into the given program as the ontic concept propagation result.

Encoding Ontology Types to Bit Vector

As a first step of encoding type constraints to a SMT problem with Bit Vector theory, Ontology qualifiers need to be encoded to Bit Vectors. The bit vector encoding of Ontology qualifiers follows the encoding specification introduced earlier in Sec. 4.3.2. The bit vector size is the number of ontic concepts appears in a given program. For ontology qualifier that represents only one ontic concept, an unit bit vector is encoded to represent the concept. For ontology qualifier represents a set of ontic concepts, the encoding is computed by the union operation on corresponding unit bit vectors.

One thing worth to be mentioned is the special treatment on polymorphic qualifer `@PolyOntology`. `@PolyOntology` is filtered out when encoding constraints, and there is no unit bit vector for `@PolyOntology`. The reason is that `@PolyOntology` is just a place holder for separating invocation contexts. Encoding constraint variables that represents the concrete invocation context is enough for modeling the corresponding SMT problem.

Tuning on Subtype Constraints

Ontology is an under-constrained system, as a valid solution for a set of constraints is not always the desired solution, i.e. the constraints are not enough to express the desired solution. For illustration, given a subtype constraint $SEQUENCE <: v_1$. TOP and $SEQUENCE$ are both valid solution for v_1. However, $SEQUENCE$ is a better solution as it reflects more useful information on v_1.

In order to guide solver to give a desired solution, instead of producing a random valid solution, subtype constraints are tuned by preferences. Generally, solutions that make more subtype variables equal to super type variables are preferred. This is because ontic concepts on base cases are desired to be propagated as much as possible. By preferring equal solutions on variables in subtype constraints, variables that connects to base cases by subtype constraints are preferred to have a solution that equals to the base case they are connected to.

Specifically, different weights are made for different compositions of variables for a subtype constraint:

- case **constant** $<:$ **variable**: Prefer *variable* equals to *constant* with weight 3.

- case **variable** <: **constant**: Prefer *variable* equals to *constant* with weight 3.
- case **variable1** <: **variable2**: Prefer *variable*1 equals to *variable*2 with weight 1.
- for all *variable* in subtype constraints: Prefer *variable* equals to BOTTOM with weight 1.

A subtype constraint between constant and variable is added more weight that prefer the variable to be equal to the constant. This is because the variable is directly "connect" with the constant, therefore this is a strong implication that suggests this variable is related to the constant. With the preference of *variable*s to be equal for subtype constraints between pure *variable*s, this ensures the propagation from ontic concept on subtype to super type variables is propagated as far as possible.

Preference on all *variable*s equal to BOTTOM has two purposes. First, for orphan variables (variables that do not connect to any constants), they prefer to have a fixed solution that reflects no informations, instead of assigning some random ontic concepts. The most proper preference for these orphan variables should be TOP. However, since in Bit Vector encoding, it does not have a first pass of recognizing which variables are orphans, prefer all *variable*s to be TOP would lost the propagation from subtype constant to super type variables (as solver would tend to give TOP solution for these super type variables base on the preference to TOP). Therefore, as a workaround, preference on BOTTOM is chosen. This still make sense based on Karl Popper's philosophy:"A theory explains everything, explains nothing". As BOTTOM means the annotated object is related to every ontic concepts, actually this BOTTOM annotation does not give any useful information. Second, preference to BOTTOM helps on restricting the propagation depth from super type constant to subtype variables. If there are too many variables on a subtype chain of a super type constant, then solver will prefer to solve these variables as BOTTOM, instead of solve them as the ontic concepts on (indirectly) connected constant.

6.4 Implementation

The Ontology type system is implemented as a pluggable type inference system on top of the Checker Framework Inference introduced in Sec. 2.2. In addition, Ontology implements a customized `FormatTranslator` and `Z3BitVectorCodec` for using Z3 BackEnd introduced in Sec. 4.3.

Overall the implementation effort of Ontology is low, totaling around 1k non-comment, non-blank lines of Java source code. The Ontology type system uses the standard type constraints generation rules in the Checker Framework Inference, only the encoding of subtype constraints is tuned by customized preference.

Ontology also provide a Python script that allows clients of Ontology to import their base cases into Ontology. Clients of Ontology only need to provide JSON files that describes their type summarization rules, and fields labeled by customized concepts, following the specific format instruction [1]. Then, for importing type summarization rules, the utility script will update Ontology with the new ontic concepts and type rules by modifying corresponding Java files and recompile the Ontology. For importing ontic concepts in fields in programs, the script will annotate fields in programs with Ontology annotations that represent corresponding concepts by using Annotation File Utilities [2].

For a given program, the produced result of Ontology is a copy of the given program that annotated by the propagated Ontology annotations. For the places where the corresponding constraint variables have a solution of TOP or BOTTOM, Ontology does not insert them into the program, as TOP and BOTTOM solution provide no useful information for building a meaningful abstraction of a given program. The annotated program follows the standard Java 8 type annotation syntax. Consequently, any standard Java parsers (e.g. JavaParser[3] or Eclipse JDT[4]) that support Java 8 syntax is able to parse the Ontology annotations in the annotated program correctly.

6.5 Experiments and Evaluations

We evaluate Ontology in two different experiments. Sec. 6.5.1 describes the evaluation of Ontology on summarizing two built-in concepts SEQUENCE and DICTIONARY, by running Ontology on 15 un-annotated real-world projects (code size range from 393 LoC to 86k LoC). Sec. 6.5.2 describes the evaluation of Ontology on propagating domain specific ontic concepts labeled in fields of programs. In this evaluation, we manually annotated two physical engine libraries by annotating the fields in libraries with 6 domain-specific ontic concepts, and then evaluating the propagation result of Ontology on these two pre-annotated libraries.

[1]See https://github.com/aas-integration/integration-test2/blob/master/map2annotation/README_ontology.txt.
[2]See https://checkerframework.org/annotation-file-utilities/.
[3]See https://github.com/javaparser.
[4]See https://www.eclipse.org/jdt/.

Benchmark	Project Size				Constraint and Slot size		
	File	Blank	Comment	Java LoC	Variable Slot	Constraint Subtype	Constraint Equality
imagej	7	110	347	393	221	126	58
FaceDetection	14	162	72	538	766	684	103
imgscalr	11	302	2186	1181	964	841	149
jump	23	770	3211	1937	1766	2404	129
jama	10	589	931	2599	4318	4706	305
jdepend	60	1501	1746	4054	2792	2852	378
exp4j	30	819	761	5129	5389	1839	1034
jdeb	73	1471	1917	5201	1829	552	195
react	63	1147	6338	10095	12890	17075	1120
jReactPhysics3D	105	3515	6251	10455	10030	15379	1241
JLargeArrays	17	882	2496	11337	15554	16670	940
jblas	74	3344	6807	11841	19330	24068	796
la4j	117	3630	5376	13480	13736	16781	854
matrix-toolkits-java	230	5306	12803	15950	15199	19185	1657
ode4j	452	21575	54317	86534	54302	70702	5173

Figure 6.3: Size of Projects, and corresponding generated constraints and slots number with running Ontology Type system.

6.5.1 Experiment on Propagating Ontology Built-in Concepts

As an initial experiment, we run Ontology on 15 un-annotated real-world projects. The purpose is evaluating Ontology on summarizing built-in concepts SEQUENCE and DICTIONARY. Fig. 6.3 shows the sizes of each project in the benchmark. The number of Java files, comments, blanks, and line of source code are computed by the `cloc` tool. The sizes of generated constraints and variable slots (constraint variables) are also shown in the table, to give a more obvious correlation between projects sizes and the number of constraints and variable slots that are generated by Checker Framework Inference.

To evaluate the effect of preference tuning, we run Ontology on the benchmark in a A/B test way. Fig. 6.6 gives the type inference result and Fig. 6.7 gives the timing result of running Ontology with/without preference tuning on the benchmark. In these two tables, the number in blue color shows the inference result without preference tunning, and the number in red color shows the inference result with preference tuning. The number of generated constraints and slots are invariants in this A/B test, and they are shown in the table for giving a more obvious correlation between running results and problem sizes.

We first discuss Fig. 6.6 to see how preference tuning influences the inference result. Then we discuss Fig. 6.7 to evaluate the timing overhead added by preference tuning.

Discussion on Fig. 6.6:Inference Result with/without Preference Tuning

To discuss Fig. 6.6, we compare related columns and draw corresponding conclusions:

Compare Variable Slots and Total Inferred:

Notice that there is a difference between the total number of generated variables slots and the total inferred solutions for these variable slots. This is because in the Checker Framework Inference level, Existential Constraints are generated for type variables. Ontology does not support Existential Constraints, therefore the variable slots in Existential Constraints are ignored.

Compare Total Inferred with/without preference setting:

An interesting result is the size of total inferred solutions are different in this A/B test. With preference setting, solver is able to give more solutions. Theoretically this does not make sense, as the size of encoded variable slots passed to solver are invariants in this A/B test. It is expected that solver gives each passed variable slot a solution. However, debugging result on the smallest project `imagej` shows that solver does not always give each passed variable slot a solution, i.e. Z3 will exclude some variable slots in the solved model. For illustration, the total encoded variable slots number for `imagej` is 158. However, without preference setting, the solved model returned by Z3 only contain solutions for 154 encoded variable slots. The reason of why Z3 excludes some variable slots in the solved model is unknown. In contrast, with preference setting, as a preference for each encode variable slot to be BOTTOM is added, Z3 returns a solved model contains solutions for all encoded variable slots.

Compare concrete inferred result with/without preference setting:

Overall, without preference setting, Ontology tends to infer most of the variable slots as TOP. In contrast, with preference setting, Ontology tends to give a "lower" solution in the type hierarchy, i.e. infer more variable slots as BOTTOM.

Regarding the result of inferring meaningful solutions (SEQUENCE, DICTIONARY), it is interesting to see without preference setting Ontology infers more variable slots with non-empty meaningful concepts than the preference setting. However, numbers does not reflect the accuracy of solutions. Therefore, as a further investigation, the annotated result by

Ontology is manually examined on a randomly selected group of projects: 2 small projects (`imagej`, `FaceDetection`), 3 medium projects (`jama`, `react`, `jReactPhysics3D`), plus the largest project `ode4j`.

The manually examine result shows, without preference setting, Ontology does not correctly infer the two built-in concepts. Most of the variable slots that inferred with an non-empty concept are orphan slots that does not connects to any constant concepts in the constraints. The inferred result of these orphan slots are just due to the random solution given by solver, as orphan slots are actually not constrained. Fig. 6.4 shows an example of the annotated result of Ontology without preference tunning. The field of type `int` in the example is just randomly inferred by Ontology with SEQUENCE concept, as the constraint variable represent this place is an orphan slot.

```
// FaceDetection/FramesDrawer.java: line 33
@Ontology(SEQUENCE) int scanningWindowSize = MIN_SIZE;
```

Figure 6.4: A code example of the annotated result of Ontology without preference tuning.

In contrast, with preference setting, although Ontology reasons about less non-empty concepts, the inferred results are meaningful. The SEQUENCE and DICTIONARY solutions inferred by Ontology are all correct in the 6 manually examined projects. Fig. 6.5 shows examples of the annotated result of Ontology with preference tuning. Ontology correctly reasons `Array` type and `List` subtypes to SEQUENCE concept, and correctly reasons `Map` type to DICTIONARY concept.

Notice the line 5 in Fig. 6.5 gives an example of reasoning field `frame` with an abstract `Collection` type to SEQUENCE. Although `Collection` is not a subtype of `List`, the inference result suggests this field is only assigned by `list` subtypes, and therefore inferring it to SEQUENCE reflects the precise semantic concept of this field.

Line 9 in Fig. 6.5 gives an example of Ontology reasoning a precise semantic meaning of a field defines an "edge adjacency list" in the physical library *jReactPhysics3D*. A text-based program analysis is very likely to reason about this field to SEQUENCE concept, as "list" is a keyword in both the code comment and the name of this field. In comparison, Ontology gives a more precise conclusion of this field: it is a DICTIONARY whose value is SEQUENCE. This conclusion may help to other program analysis tools to better understand what is an "adjacency list" in this physical library.

In conclusion, the A/B test result suggests that Ontology requires preference tuning to give a meaningful inference result, as Ontology type system itself is under-constrained.

```
// Jama/LUDecomposition.java: line 39
int @Ontology(SEQUENCE) [] piv;

// FaceDetection/FramesDrawer.java: line 33
@Ontology(SEQUENCE) Collection<Frame> frames = new ArrayList<>();

// jreactphysics3d/ConvexMeshShape.java: line 71

// Adjacency list representing the edges of the mesh
@Ontology(DICTIONARY)
Map<Integer, @Ontology(SEQUENCE) List<Integer>> edgeAdjList;
```

Figure 6.5: Code examples of the annotated result of Ontology with preference tuning.

Discussion on Fig. 6.7: *Timing result with/without Preference Tuning*

The timing result shows preference tuning almost has no influence on encoding time, and doesn't hurt solving performance a lot for small projects($<$ 10k LoC), but do add non-trivial solving time overhead for large projects ($>$ 10k LoC). On average, for the 8 projects which size is less than 10k LoC, preference tuning add on average 2.66 times overhead is added for solving time. However, for projects more than 10k LoC, the preference tuning add a lot time overhead. For the 7 large projects, on average preference tuning adds 9.2 times overhead on solving time. For the worst case, preference add 22 times overhead for solving (project `jblas`).

6.5.2 Experiment on Propagating Domain-specific Concepts

As an advanced experiment, we manually annotate two physical libraries: `jReactPhysics3D` and `ode4j` with six ontic concepts that frequently appears in these two libraries: COORDINATES, VELOCITY, FORCE, TORQUE, AXIS, and ANCHOR. We manually annotate several fields in these two libraries with these 6 ontic concept as the simulated pre-knowledge input for Ontology. The annotation on each field is decided by examining the field name or reading the code comments. For example, if a field is named with "xxxVelocity", then we annotated it by `@Ontology(VELOCITY)`. After the manual annotating work, we run Ontology on these annotated libraries, to see how well it propagates these ontic concepts. Fig. 6.8 shows the running result. The number in blue color is the amount

Project Name	Variable Slots	Total Inferred		TOP		BOTTOM		SEQUENCE		DICTIONARY	
imagej	221	154	158	141	52	11	99	-	3	2	4
FaceDetection	766	584	613	570	222	-	377	14	12	-	2
imgscalr	964	710	756	649	417	-	333	26	6	35	-
jump	1766	1417	1565	1401	975	-	538	16	51	-	1
jama	4318	3101	3568	3083	2165	-	1278	15	125	3	-
jdepend	2792	2198	2416	2147	1104	-	1163	33	134	18	15
exp4j	5389	2600	2843	2598	1404	-	1426	2	7	-	6
jdeb	1829	728	795	725	274	-	488	3	26	-	7
react	12890	10318	10984	9644	3174	-	7559	674	242	-	9
jReactPhysics3D	10030	8631	9014	8582	2459	-	6267	36	266	13	22
JLargeArrays	15554	11655	12771	11552	6779	1	5363	102	629	-	-
jblas	19330	15970	17159	15382	4999	-	11274	588	886	-	-
la4j	13736	10093	11226	8966	6203	2	4818	1125	205	-	-
matrix-toolkits-java	15199	11645	12456	11296	4433	2	7037	347	979	-	7
ode4j	54308	44780	46788	41507	12718	2	32777	3271	1289	-	4

Figure 6.6: Type inference result of running Ontology with/without preference tuning.

Project Name	Var. Slots	Cons.	Prefer. Cons.	Encoding Time (ms)			Solving Time (ms)		
				No P	P	over head	No P	P	over head
imagej	221	185	347	19	54	1.84	12	170	13.17
FaceDetection	766	790	1450	51	49	-0.04	110	167	0.52
imgscalr	964	994	1805	55	63	0.15	106	240	1.26
jump	1766	2541	4170	98	84	-0.14	1039	1156	0.11
jama	4318	5014	9024	90	107	0.19	860	1629	0.89
jdepend	2792	3252	5644	82	102	0.24	511	1070	1.09
exp4j	5389	2939	7228	94	94	0.0	137	486	2.55
jdeb	1829	754	2381	59	64	0.08	58	157	1.71
react	12890	18244	29965	243	255	0.05	5269	16013	2.04
jReactPhysics3D	10030	16656	25409	271	246	-0.09	3801	13028	2.43
JLargeArrays	15554	17663	32224	282	307	0.09	2455	38528	14.69
jblas	19330	24889	43398	317	400	0.26	3574	84380	22.61
la4j	13736	17863	30517	256	295	0.15	3660	21059	4.75
matrix-toolkits-java	15199	20951	34384	271	324	0.2	4221	40189	8.52
ode4j	54308	76196	125004	715	840	0.17	41407	431235	9.41

Figure 6.7: Timing result of running Ontology with (P)/without (No P) preference tuning. Text in blue color represents the result of running Ontology without preference, and text in red color represents the result of running Ontology with preference. Run on Mac OS, with 2.3 GHZ Intel Core i7 processor, 16 GB memory.

of pre-annotated annotations, and the number in red color is the amount of propagated annotations.

The result shows Ontology be able to propagate 3.4 annotations on per pre-given annotation. We manually examine the running result, and find all of the propagated annotations correctly reflects the related concepts of the annotated object. Fig. 6.9 shows three representative example of propagation results.

```
 1 // jReactPhysics3D/BoxShape.java: line 46
 2 // Pre-annotated
 3 // Extent sizes of the box in the x, y and z direction
 4 @Ontology(values={COORDINATES}) Vector3 extent;
 5 //...
 6 // Propagated result: line 64
 7  public @Ontology(COORDINATES) Vector3 getExtent() {
 8         return extent;
 9  }
10
11  // ode4j/DxJointAMotor.java: line 51
12  @Ontology(values={AXIS})
13  DVector3 @Ontology(SEQUENCE}) [] axis;
14
15  // ode4j/DxJoint.java: line 466
16  void setBall(@Ontology(ANCHOR) DVector3 anchor1,
17  @Ontology(ANCHOR) DVector3 anchor2) { ... }
```

Figure 6.9: Code examples of the annotated result of Ontology with preference tuning.

For code examples in Fig. 6.9, the first example on line 7 suggests Ontology is able to propagate the knowledge obtained from fields' code comments to APIs that used these fields. The COORDINATE concept annotated on line 4 is a manual annotation result by reading the code comments of this `extent` field (as the comment include keywords "x, y, and z direction""). The `getExtent` method does not have any Javadoc and comments. Ontology helps to propagate the COORDINATE concept analyzed from field `extent` to the API `getExtent` which lacks of proper code comment that document this knowledge.

The second code example on line 12-13 suggests Ontology is able to combine domain-specific ontic concept and built-in ontic concept to produce a meaningful composition of concepts. Text-based analysis may simply concludes field `axis` is related to AXIS concept

Project Name	Total		COOR -DINATES		VELOCITY		FORCE		TORQUE		AXIS		ANCHOR	
	Pre.	Inf.	Pre.	Inf.	Pre.	Inf.	Pre.	Inf.	Pre.	Inf.	Pre.	Inf.	Pre.	Inf.
jReactPhysics3D	30	77	13	45	2	4	14	24	1	4	-	-	-	-
ode4j	46	197	9	48	2	14	4	32	2	8	17	53	12	38

Figure 6.8: Inference result of Ontology type system on propagating pre-annotated ontic concepts.

by name analysis. However, Ontology gives a more precise conclusion of field `axis`: it is a SEQUENCE of AXIS.

The third code example on line 16-17 suggests Ontology verifies the method parameters `anchor1` and `anchor2` are indeed relate to ANCHOR concept. This is because through the type inference of Ontology, only fields/variables related to `ANCHOR` concepts is passed to the method `setBall`. Therefore, the inference result concludes these two method parameters are related `ANCHOR` concept with high confidence.

6.6 Future Work

Similar to the preference tuning in Dataflow, the current preference tuning for Ontology is also an "equality" preference, and is unable to guide solver gives desired solutions to variables on intersections of base cases. A more desired preference tuning would be a "least upper bound" preference tuning, i.e. a preference that prefers each bit vector should have as most bits set to 1 as possible. For example, given a constraint variable that is constrained as the super type of both base case $\{SEQUENCE, FORCE\}$ and base case $\{SEQUENCE, VELOCITY\}$, a desired preference tuning is to prefer the bit vector that represents this variable to have the most bits set to 1, which will guide solver gives $SEQUENCE$ as the solution to this variable. A brute force approach is to prefer each bit vector to include every unit bit vector as its subset. However, this enumeration-based approach requires n preference constraints for each bit vector for a power-set-like type hierarchy which the universe set size is n. Consequently, too many preference constraints are generated, which slowdowns the solving time to an unreasonable range. Finding an efficient encoding for expressing this "least upper bound" would be an interesting future work.

In addition, we would like to enrich the type qualifier hierarchy of Ontology to express more relationships between Ontology qualifiers. For example, we would like to mark elements that will flow into a SEQUENCE, and elements that come from a SEQUENCE. This requires a more expressive type qualifier hierarchy, or have multiple type qualifier hierarchies to cooperate together.

Moreover, as Sec. 6.2.2 indicates, Ontology currently cannot infer @*PolyOntology*. Requiring clients of Ontology to manually insert @*PolyOntology* annotations into programs is a non-trivial task. An interesting future work would be investigating on how to enable Ontology to infer @*PolyOntology* for programs.

6.7 Conclusion

This chapter presents a novel type system named Ontology that reasons about a coarse abstraction for a given program based on ontic concepts. As an infrastructure of ontic concept propagation, Ontology provides two built-in ontic concepts SEQUENCE, and DICTIONARY. In addition, clients of Ontology can add additional ontic concepts by providing customized type summarization rules or a ground truth set that annotated fields in programs with customized ontic concepts.

In a case study of running Ontology on 15 real world scientific libraries(project size range from 393 to 86k LoC), it summarizes 4973 built-in ontic concepts, and propagates 274 domain-specific concepts in two pre-annotated physical libraries. We manually examine the inference result, and discuss several interesting cases that Ontology might be helpful for other program analysis tools to better understand programs. The experimental result suggests Ontology is able to produce meaningful high-level program abstractions based on ontic concepts.

Chapter 7

Conclusion

This thesis presents work on three pluggable type systems for helping programmers and other program analysis tools to better understand programs.

We present the EOF Value Checker, which prevents unsafe end-of-file (EOF) value comparisons that can lead infinite loops in programs. EOF Value Checker examines additional safety-related properties on integer types, and statically ensures no FIO-08J rule violations appear in a given program.

We also present a novel type system named Ontology, which reasons about a coarse abstraction for a given program based on ontic concepts. An ontic concept can be a type summarization rule that groups similar Java types to a general ontic concept, or it can come from pre-annotated fields in programs. Ontology provides two built-in ontic concepts. The first groups Java `Array` type, `List`, and subtypes of `List` as SEQUENCE concept. And the second groups Java `Dictionary`, `Map`, and their subtypes to DICTIONARY concept. Clients of Ontology can also provide customized ontic concepts by providing JSON files that indicates their type summarization rules and annotated fields. Ontology is able to produce reasonable semantic abstractions for a given program, so that the produced abstraction may facilitate other program analysis tools on their tasks.

In addition, we present our work on extending the *Type Constraint Solver* from supporting only Max-SAT solvers to also supporting Max-SMT solvers. This is a foundational work for the new SMT approach based on Bit Vector theory for solving the type constraints of Ontology type system.

We also apply the new SMT encoding with Bit Vector theory to Dataflow type system. The SMT encoding approach makes Dataflow be able to infer more run-time types in programs, and also be possible to support partially annotated programs.

For experimentation, we evaluate EOF Value Checker on 35 real world projects, and evaluate Ontology and Dataflow type systems on 15 scientific libraries. Through analysis of the experimentation results, we conclude that pluggable type systems can provide confidence on preventing formalized vulnerabilities, and be able to infer high-level abstractions for programs.

References

[1] Boolean satisfiability problem. Wikipedia. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem; accessed 02.03.2018.

[2] Conjunctive normal form. Wikipedia. https://en.wikipedia.org/wiki/Conjunctive_normal_form; accessed 02.03.2018.

[3] Maximum satisfiability problem. Wikipedia. https://en.wikipedia.org/wiki/Maximum_satisfiability_problem; accessed 02.03.2018.

[4] Satisfiability modulo theories. Wikipedia. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories; accessed 02.03.2018.

[5] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings On Production Software. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[6] Roberto Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design.* Birkhauser, January 1995.

[7] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *Software Engineering in Practice Track, International Conference on Software Engineering (ICSE)*, May 2011.

[8] Distinguish between characters or bytes read from a stream and -1. In [18] and available online, November 2017. https://wiki.sei.cmu.edu/confluence/display/java/FIO08-J.+Distinguish+between+characters+or+bytes+read+from+a+stream+and+-1; accessed 25.11.2017.

[9] C. Hoare. Null references: The billion dollar mistake. abstract of QCon London Keynote, available online, 2009. qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake, OPTannote = .

[10] S. Elbaum K. T. Stolee and D. Dobos. Solving the search for source code. In *ACM Trans. Softw. Eng. Methodol.*, 2014.

[11] Jianchu Li. A General Pluggable Type Inference Framework and its use for Data-flow Analysis. available online. https://uwspace.uwaterloo.ca/bitstream/handle/10012/11771/Li_Jianchu.pdf?sequence=1; accessed 27.02.2018.

[12] A. Mahmoud and G. Bradshaw. Estimating semantic relatedness in source code. In *ACM Trans. Softw. Eng. Methodol.*, 2015.

[13] Telmo Luis Correa Jr. Jeff H. Perkins Matthew M. Papi, Mahmood Ali and Michael D. Ernst. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis*, 2008.

[14] A. K. Dwivedi P. Pradhan and S. K. Rath. Detection of design pattern using graph isomorphism and normalized cross correlation. In *International Conference on Contemporary Computing*, 2015.

[15] Mikael Rittri. Using types as search keys in function libraries. In *the fourth international conference on Functional programming languages and computer architecture*, June 1989.

[16] Colin Runciman and Ian Toyn. Retrieving reusable software components by polymorphic type. In *Journal of Functional Programming*, April 1991.

[17] F. Long S. Sidiroglou-Douskos, E. Lahtinen and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[18] David Svoboda, Dean F. Sutherland, Robert C. Seacord, Dhruv Mohindra, and Fred Long. *The CERT Oracle Secure Coding Standard for Java.* Addison-Wesley Professional, September 2011.

[19] M. Vechev V. Raychev and A. Krause. Predicting program properties from big code. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[20] M. D. Ernst W. Dietl and P. Mller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[21] S. Drossopoulou W. Dietl and P. Mller. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[22] C. Le Goues Y. Ke, K. T. Stolee and Y. Brun. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering*, 2015.