A General Pluggable Type Inference Framework and its use for Data-flow Analysis

by

Jianchu Li

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, 2017

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

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

Abstract

Java's pluggable type systems provide valuable compile-time guarantees, but annotating the program with pluggable types can be a significant burden on programmers. Checker Framework Inference, a framework that aims to provide a constraint-based type inference for pluggable types, can generate type constraints over the occurrence of type qualifier of expressions according to type rules. However, there is no efficient approach to solve type constraints generated by Checker Framework Inference.

This thesis presents a system called *Type Constraint Solver* that can solve the type constraint by encoding the constraint as a Max-SAT problem and in the LogiQL language. The system takes advantage of existing Max-SAT solvers and LogicBlox to solve the corresponding forms, and gets the concrete pluggable type qualifiers for program expressions. *Type Constraint Solver* provides options to separate constraints into groups and solve them in parallel. It also has extendability that can be easily extended with custom encoding logic. We developed a pluggable type system called *Dataflow Type System* on top of Checker Framework Inference to verify the functionality of *Type Constraint Solver*. The type system and its inference can perform data-flow analysis by inferring all possible run-time Java types of return types, parameters, fields, and variables at compile time. We applied Checker Framework Inference to six real-world applications of up to 39kLOC with *Dataflow Type System* and *OsTrusted Type system* resulting approximately 58,000 type constraints. We used our tool to solve these type constraints and analyzed the experimentation statistics. We manually examined the inference result and found that *Type Constraint Solver* is able to automatically infer the expected type qualifiers for benchmarks. Inferring the largest application with fastest inference options took about 10 seconds on average, and approximately 23,000 type qualifiers were inferred. These results suggest that our system can efficiently give correct solution for type constraints.

Acknowledgements

I would like to thank my supervisor Professor Werner M. Dietl for his support and guidance. I thank Jeff Luo for his very useful feedbacks on my thesis. I would also like to thank my readers, Professor Ondřej Lhoták and Professor Derek Rayside for their time, feedback, and questions. I am thankful for all the members of Professor Dietl's research group who made the work environment friendly and encouraging.

Dedication

This is dedicated to my parents who have always been there for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reliability is an important facet of software quality. In order to make software more reliable, many elaborate and expressive approaches have been come up with and implemented. Meanwhile, as one of the most popular programming languages in the world, Java has been widely used in many places. From Android applications to web applications, from scientific applications to financial applications like electronic trading systems, from games like Minecraft to desktop applications like Eclipse and IntelliJ etc.[10], Java is almost everywhere in the real world. Because of the popularity of Java, the reliability of Java software has been becoming more and more important.

People put lots of effort in developing different approaches and techniques in order to make Java programs reliable. We want to have approaches that have both high practicality and also can make the guarantees as much as possible. However, in many cases they cannot be both achieved, and they are even in a negative correlation sometimes. For example, code reviews is a very common way to improve code quality and find potential bugs. An experienced senior programmer can always give us wise suggestions, but reading all of the codes line by line, compiling and running them in the mind would not be very practical, and lots of run time errors are unreadable to programmers. Compare to code review, software testing is more practical. There are different kinds of testing method, for example, static and dynamic testing, white and black box testing etc. Software testing can be performed in different levels, and also lots of testing techniques has been developed. Software testing is practical and it can make sure the code works the way that it's supposed to work, however, in some cases it cannot provide enough guarantees that let people trust the software. For example, if we write a program that counts from 0 to an arbitrarily large natural number n, and we want to test the program. The normal way is we come up with some sufficiently large integer numbers, and run the program with the numbers, for instance, 1000, and we look to see whether the program can list $1, 2, 3...1000$. If it shows those results, the test passes, otherwise, it fails. Since we cannot enumerate all possible integer numbers, so we cannot make sure that the program would still work when the integer n is larger than the largest number we have tested. There is another technique called formal verification, which allows you to prove mathematically that the underlying algorithm is correct, and offers much more guarantees to programmers. For above case, we can prove the theorem, for any arbitrarily large natural number n, it is possible to count from 0 to n by induction, so that make sure our algorithm is correct. But if the logic of the program is too complicated, proving theorems would be a huge workload for programmers.

Type system is another way to prevent the occurrence of execution errors. Type system defines interfaces between different parts of a computer program, and then checks that the parts have been connected in a consistent way. If the program passes type checking, the properties that defined by type rules can be hold. If the type system is properly developed and formalized, in other word, as long as we prove the type system is sound, it can provide mathematical guarantees. Type system can also be practical, since the implementation of type system is in language level and not program specific.

The Java programming language has a strong type system that can be used to enforce useful program properties. The basic Java type check can prevent many errors. For example,

```
int i = "Hello";
```

we will get an incompatible types compile time error if we compile the above code. However, Java built-in type system cannot express enough properties in compile time. For example:

```
int getCounter(Input input) {
  return input.getCounter();
}

int counter = getCounter(null);
```

In above case, we will get a null pointer exception in run time, however the Java compiler cannot catch it in compile time. Many important properties just like nullness of object can be hard to encode as standard types, and it may be difficult to incorporate new properties into the type hierarchy of an existing program.

To address this problem, *Checker Framework* [7, 27] has been presented. *Checker Framework* is a tool that can support adding pluggable type systems to the Java language, and let programmers run an additional type-checker as a plug-in to the javac compiler. A type system designer can use the *Checker Framework* to create a type checker by defining type qualifiers and their semantics, and then enforce the semantics at compile time. Programmers can write the type qualifiers in their programs and use the type checker to detect or prevent errors[7].

For example, we can use *Nullness Checker* to find the null pointer problem in above code. We first annotate the parameter of method getCounter() with *@NonNull* annotation to indicate that the parameter cannot be null:

```
int getCounter(@NonNull Input input) {
  return input.getCounter();
}

int counter = getCounter(null);
```

Then if we compile the program with *NullNess Checker*, we will get following error:

```
error: [argument.type.incompatible] incompatible types in argument
    int counter = getCounter(null);
  found    : null
  required: @NonNull Input
```

Checker Framework not only is an expressive tool allows type system designer to plug their own type systems into Java language, but also introduces a set of checkers that can provide valuable compile-time guarantees. However, programmers must annotate the program with the pluggable types, and that can be a significant burden. Because of the vast quantity of legacy code, programmers need to spend a long time on studying the legacy code before he or she understands the code well and annotates it.

Type inference is an approach that can automatically determine the types for programs such that reduces the burden for programmers. To infer pluggable types that defines on top on *Checker Framework*, *Checker Framework Inference*, a type inference framework for *Checker Framework*, has been presented [1].

Checker Framework Inference is based on static analysis, implementing a constraint-based type inference approach. It can generate different kinds of type constraint over program locations according to the type rules to express the relationship among the type qualifiers that should be annotated in these locations. If we can find a set of type qualifier that can satisfy the type constraints, we will determine the type qualifiers for those locations. The details about *Checker Framework Inference* are introduced in Section 2.1.

1.1 Motivation

Inferring pluggable type qualifiers would help programmers to reduce their workload and make it easier to use type checkers. *Checker Framework Inference* implements a constraint-based type inference, which works by constructing a system of constraints that express relationships between the pluggable types over different parts of the program. However, there isn't an effective way to solve the constrains from any pluggable type system and get concrete type qualifiers. The inference framework only introduces a few constraint solvers for some specific type systems, but we would like to have a general approach to solve type constraint from arbitrary type systems.

In this research, we wanted to find a solution for solving the constraints by encoding the constraint to some other forms, then solves the new form through its corresponding solver, decoding the result, and getting the solution of the constraints. In practice, we not only looked for a constraint encoding strategy, but also tried building a constraint solving system that can be easily extended by adding new encoding approaches. In order to verify the correctness of our constraint solving system, we also wanted to build an inferable type system, run the inference tool on it, and check the inferred results.

1.2 Approach

In this research, we propose two different encoding strategies for type constraints generated by *Checker Framework Inference*: *Max-SAT* and *LogiQL*. The former one encodes the constraint as a weighted Max-SAT problem such that each constraint would become to a set of Boolean clauses, and uses an existing solver to solve the Boolean formula. The latter one converts the constraint into statements of a Datalog based language *LogiQL*, and processes the statements by *LogicBlox*, a database running with *LogiQL*.

We built an infrastructure level system, which integrates above two encodings in Java with proper usage of object oriented programming concepts, design patterns, and graph related algorithms. The system can solve type constraints from arbitrary type system and get the concrete type qualifier for each location.

In order to validate the correctness of the inference tool, we built a pluggable type system *Dataflow Type System* on the top of *Checker Framework* and *Checker Framework Inference*. *Dataflow Type System* and its inference can infer all possible run-time Java types of return types, parameters, fields, and variables at compile time.

For case studies, we ran our tools on open source software projects with different sizes. We collected different aspects of performance related statistics to demonstrate the performance of our tools.

1.3 Thesis Contributions

The contribution points of this thesis are as follows:

First, a type constraint encoding approach based on Max-SAT problem, which can convert the constraint into boolean formula and be solved by Max-SAT solver.

Second, another LogicBlox based type constraint encoding. It encodes constraints as LogiQL language and solves them through LogicBlox database.

Third, designs and implements *Type Constraint Solver* system that can solve type constraint generated from arbitrary type systems, and integrates above two encodings into the system. We made the infrastructure of *Type Constraint Solver* can be easily extended with new encoding strategies or custom constraint solver for particular type systems with special requirement.

Fourth, designs and implements inferable *Dataflow type system*. The type system and its inference can perform data-flow analysis[1] for Java, and also help to validate the correctness of the type constraint solver system.

Fifth, it evaluates the performance of proposed type constraint solver system by inferring Java projects in different sizes.

1.4 Thesis Organization

Chapter 2 discusses the background knowledge of *Checker Framework Inference*, *Max-SAT* problem and *LogicBlox*, and also presents other type inference approaches that related

[1] Data-flow analysis that the type system focuses on is the type of each allocation in Java, see Chapter 4 for more details.

to this thesis. Chapter 3 introduces *Type Constraint Solver* in detail including the structure of the system and different constraint encodings. Chapter 4 presents the logic of *Dataflow Type System*. Chapter 5 describes our implementation and our experience with the tools. Chapter 6 discusses the possible future works. Finally, a summary of the work and conclusion are give in Chapter 7.

1.5 Funding

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada. This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force and the Defense Advanced Research Projects Agency (DARPA).

Chapter 2

Background and Related Work

This Chapter explains the background knowledge that is referred by and used in this research. Section 2.1 introduces the basic logic in *Check Framework Inference* including the construction of constraint variables and constraints over the variables. Section 2.2 and Section 2.3 talks about the *Max-SAT* problem and *LogiQL* languages respectively. Prior works are discussed in Section 2.4.

2.1 Background on Checker Framework Inference

This section introduces the basics about *Checker Framework Inference* and an existing inferable type system called *OsTrusted Type System*.

2.1.1 Checker Framework Inference

Checker Framework Inference is a framework that can perform type constraint generation for the pluggable type system defined in *Checker Framework*. Since Java SE 8 release, Java annotations can be used as pluggable types, and *Checker Framework* is a pluggable type checking framework that allows developers to define their own type system and plug it into the normal Java language to perform some additional compile time check [7, 27]. It uses the *annotation* in Java language to represent the type qualifier [8]. One example is *Nullness Checker*, users can annotate the program with Nullness type system qualifier to enforce some objects be non-null in order to avoid a *NullPointerException* in run time.

```
@NonNull  Object obj; // obj can never be null
```

If we assign a **null** literal to obj, and compile the code, *Checker Framework* will report an error says the null type is incompatible with the type of obj.

As we mentioned in Chapter 1, the pluggable type systems provide valuable compile-time guarantees, but it also brings overhead: the programmer must annotate the program with the pluggable type qualifiers. Annotating new code would cost much time for programmers, and annotating legacy code would be a significant burden on programmers because they have to spend a long time on studying the legacy code before they can know how to annotate the program.

The motivation of *Check Framework Inference* is helping programmer to reduce the burden of annotating the program by automatically determining types for some or all parts of the program.

The strategy that *Check Framework Inference* uses is constraint-based global type inference. It takes a program as input, and creates a set of type variables and type constrains.

Tunable Static Inference for Generic Universe Types [18] describes the generation of type variables and type constraints in detail. The rest of this section summarizes the approach and explain the connection between *Check Framework Inference* and our work.

For a given program, we would like to infer pluggable type qualifiers for different locations. A *constraint variable* represents the pluggable type qualifier for the occurrence of a particular expression, and we use it as a placeholder for the type qualifier. In *Check Framework Inference* implementation, it uses the word *slot* to refer constraint variable. For each position where a pluggable type qualifier may occur, *Check Framework Inference* introduces a constraint variable for the type qualifier annotated in that location. And our eventually goal is to decide the concrete type qualifiers for these slots.

With constraint variable generated, the system generates a set of *type constraint.* type constraint is a judgment over class, class members, and expressions according to type and constraint generation rules.

Below is the list of the five kinds of type constraint.

- *Subtype constraint* ($u_1 <: u_2$): A subtype constraint enforces that u_1 will be assigned an type qualifier that is a subtype of the type qualifier assigned to u_2.
- *Adaptation constraint* ($u_1 \triangleright u_2 = \alpha_3$): An adaptation constraint ensures that the viewpoint adaptation of variable u_2 from the viewpoint expressed by u_1 results in α_3. α_3 is a constant type qualifier in pluggable type system.

- *Equality constraint* ($u_1 = u_2$): An equality constraint ensures that two qualifiers are the same.

- *Inequality constraint* ($u_1 \neq u_2$): An equality constraint ensures that two qualifiers differ.

- *Comparable constraint* ($u_1 <:> u_2$): A comparable constraint express that two qualifiers are not incompatible, that is, one could be subtype of the other.

- *Preference constraint* ($u_1 \approx u_2$): An equality constraint indicates that we prefer the two qualifiers equal to each other.

By default, *Checker Framework Inference* generates subtype constraint and equality constraint for any type systems, and it can generate other kinds of constraint over the constraint variables according to the constraint generation rules defined in the particular type system. One scenario of equality constraint generation is when method overriding happens. If method m_1 overrides the method m_2 in a superclass, the parameters of these two methods should be consistent. So in that case, one or a set of equality constraint will be generated between the parameters in m_1 and m_2. Figure 2.1, constraint generation rules for method declaration, is cited from [18], and the set Σ_2 defines the equality constraint we mentioned above.

$$\frac{\begin{array}{c} env(\Gamma, \overline{TP}, \overline{T\,pid}) = \Gamma' \qquad \Gamma' \vdash \overline{T\,pid} : \Sigma_0 \\ \Gamma' \vdash e : T, \Sigma_1 \qquad overriding(\Gamma', m) = \Sigma_2 \\ \Gamma' \vdash bounds(\overline{TP}), T_r\ OK : \Sigma_3 \\ \Gamma' \vdash <: T_r : \Sigma_4 \end{array}}{\Gamma \vdash p\langle\overline{TP}\rangle T_r\ m(\overline{T\,pid})\ e : \cup_{i=4}^{i=0}\Sigma_i}$$

Figure 2.1: Constraint Generation Rules for Method Declaration introduced in [18]

A more common example is subtype constraint, and the constraint will be generated when subtype relationship is present in program. Figure 2.2 shows some subtype constraint generation rules from [18]. For the first rule, if type qualifier u is the subtype of another qualifier u', and the parameters for class C are same, then a subtype constraint Σ will be generated over the subtype relation $u\,C\langle\overline{T}\rangle <: u'\,C\langle\overline{T'}\rangle$.

Users can build their own type systems and define the type rules and constraint generation rules by creating a new inferable type checker on the top of *Checker Framework*

$$\frac{\Sigma = \left\{u <: u', \overline{T} = \overline{T'}\right\}}{\Gamma \vdash u\,C\langle\overline{T}\rangle <: u'\,C\langle\overline{T'}\rangle : \Sigma} \qquad \frac{\begin{array}{c}\Gamma \vdash T <: T_1 : \Sigma_1 \\ \Gamma \vdash T_1 <: T' : \Sigma_2\end{array}}{\Gamma \vdash T <: T' : \Sigma_1 \cup \Sigma_2}$$

Figure 2.2: Partial Subtype Constraint Generation Rules introduced in [18]

Inference. *Generic Universe Types* [18] is a good example to illustrate how viewpoint constraint and comparable constraint are generated. Another example is *Dataflow* type system, which will be introduced in Chapter 4.

Checker Framework Inference implements these rules, by creating type constraints during the visitation of abstract syntax tree. When framework traverses AST of the given program, if the location needs to be annotated, the framework will generate a constraint variable for the location, and then do a deep traversal. The constraint variable would be set to a constant of a certain type qualifier if the programmer has already manually annotated the location, or it can be forcibly set to a particular qualifier by type rules. In traversal process, if the system finds a scenario that satisfies the constraint generation rules, a corresponding type constraint will be generated.

Here is an example:

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

In above snippet of code, the return type of method foo is Object, and the two branches return new object and integer literal respectively. There will be three newly generated type variables: v_1 for the method type, and v_2 and v_3 for the expressions in return statement. And two subtype constraints will be generated:

$$v_2 <: v_1$$

$$v_3 <: v_1$$

The expected solution for above constraints depends on the underlying type system. For *Dataflow* type system, the concrete type qualifiers for v_1, v_2, and v_3 should be:

$$\begin{aligned} v_1 &= \text{@DataFlow(typeNames=''Object'', ''Integer'')} \\ v_2 &= \text{@DataFlow(typeNames=''Object'')} \\ v_3 &= \text{@DataFlow(typeNames=''Integer'')} \end{aligned}$$

Chapter 4 will discusses the above example in detail.

Once *Checker Framework Inference* generates a set of type constraint according to constraint generation rules, we need to solve them in order to get the concrete pluggable type qualifiers for the type variables. Although there are few type constraint solvers that can solve constraints from some particular type systems, the usability of the solvers are limited, since those target type systems are less sophisticated. So we would like to have a general solution that can solve constrains from arbitrary type systems.

Chapter 3 introduces a general type constraint solving system that can take the type constrains as input, solve the constraints, and return the type qualifier for each type variable. The system solves the constraints by encoding the type constraints into *Max-SAT* problem or statements of *LogiQL* language.

2.1.2 OsTrusted Type System

OsTrusted Type System is a simple inferable type system developed by *Checker Framework Inference* team. The type system has a two-qualifier hierarchy: **OsTrusted** and **OsUnTrusted**, and a polymorphic qualifier for **PolymorphicQualifier**. **OsUnTrusted** is supertype of **OsTrusted**, Figure 2.3 shows the hierachy :

OsTrusted type qualifier indicates that the annotated string is trusted by operating system, and operating system doesn't trust all other strings. The only type rule in OsTrusted Type System is in string concatenation: only the concatenation with two OsTrusted strings can result a OsTrusted string, and any other concatenation results in OsUnTrusted.

OsTrusted type system is used for making string operation more safe. Programmers can annotated the OsTrusted type qualifier for the string they trust. If some system execution

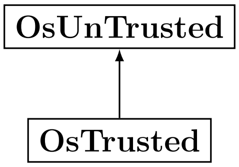

Figure 2.3: Type Hierarchy of OsTrusted Type System

calls are invoked with the string without OsTrusted type qualifier, then we can stop the invocation so that some attacks like SQL injection can be prevented.

In this thesis, we tested the type constraint solver introduced in Chapter 3 on both OsTrusted Type System and Dataflow Type System that will be discussed in Chapter 4. The statistics of the experimentation result is showed in Chapter 5.

2.2 Background on Max-SAT

Maximum satisfiability problem (Max-SAT) [6] is a generalization of the Boolean satisfiability problem (SAT). It is the problem of determining the maximum number of clauses, of a given Boolean formula in conjunctive normal form, that can be made true by an assignment of truth values to the variables of the formula.

For example, given two Boolean variables x, y and following formula:

$$(x \vee y) \wedge (\neg x \vee \neg y)$$

we know that for any truth assignment that can make the exclusive or of x and y be true, the assignment can also make the given formula be true.

But for the formula:

$$(x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$$

no matter which truth values are assigned to the two variables, at least one of the four clauses will be false. So for normal SAT problem, the above predicate is unsolvable. However, three clauses of the formula can be true with any truth assignment. Then the Max-SAT solution for the above clause would be any truth assignment for x and y, and the maximum number of clauses that can be made true is three.

According to computational complexity theory, the Max-SAT problem is a NP-hard problem[25].

Many solvers for Max-SAT have been developed during recent years. In our research, we use two of them: *sat4j*[15] and *lingeling*[5].

sat4j is a Java library for solving SAT and Max-SAT problems. Every clause is represented by an integer vector, and the variables is represented by integer values starting from 1. Negation of a variable is a negative number. The output is an array of integer. The sign of integer values indicates whether we should assign true or false to the variable.

lingeling is a modern SAT solver written in C. The input of it is a file in CNF format. For example, the clause for above formula is written in the format:

```
p cnf 2 4
1 2 0
1 −2 0
−1 2 0
−1 −2 0
```

First line is problem line. It begins with a lower case ”p” followed by a space, followed by the problem type, which for CNF files is ”cnf”, followed by the number of variables followed by the number of clauses. Starting from line 2, each line is a clause, and each variable is represented by a integer value. We use 1 and 2 to represent x and y respectively. Number 0 is the termination symbol of each clause[2]. The content of the output is same to *sat4j*'s, and the tool would just print the result out.

2.3 Background on LogiQL

LogiQL is a declarative logic programming language developed by LogicBlox, Inc, to harness the power of first-order logic to support access to databases. It has been influenced by two other logic languages, Prolog and Datalog[23].

LogiQL is a powerful language, and we used the three most significant features in *LogiQL*: *entity*, *functional predicate*, and *declaring derivation rules*. The rest of this section will introduce these three features in detail.

The definition of *Entity* is a predicate that asserts the existence of a set of elements in the problem domain that the program is modeling. Roughly speaking, *entity* is very similar to the concept *class* in object oriented programming language. It is a abstract concept, which you describe with one or more values. For example, if we want to construct a "people" entity in *LogiQL*, we can use following syntax:

```
People(p) -> .
```

The above statements declares a "people" entity. A declaration consists of two parts, separated by a right arrow. On the left is a predicate name, giving the name of entity type, *People*. A predicate denotes a named collection of facts. In this case, the facts indicate the people we refer to in our program[23]. On the right is just a single period, which means there is no information about how people entity is represented in the computer. So in that case the system would just create some internal representation automatically. So if you want to provide some ways for people entity to identify the people instead of using the default identifier, you can define a key together with the entity:

```
People(p), hasPeopleName(p:s) -> string(s).
```

The comma in the left denoting the logical *and* operator. The hasPeopleName is a *refmode* predicate that indicate for each people p, it has a key s for it. And the right of the arrow, *string(s)* indicates the key s is represented in the computer as a string.

Now that we have the *People* entity, and we can add some People facts to the system. We can do that by following syntax:

```
+People(p), +hasPeopleName(p:"Jason").
```

In above statement, the plus sign before the entity name is called a delta modifier. And this modifier indicates that the denoted fact should be added to the set of asserted fasts, and the key for this new *People* fact is a string "Jason".

Let's talk about *functional predicate* next. A *functional predicate* can express the relationship among *entities*. For example, besides *People* entity, we also have an entity called *Address*:

```
Address(a), hasAddressName(a:s) -> string(s).
```

Assume that each people have one address to live, and this association between people and their address is said to be *functional*, because address can be determined from the people. We can use following syntax to define a *functional predicate*, which would indicate the relationship between entities *People* and *Address*.

```
addressOf[p] = a -> People(p), Address(a).
```

The above statement can be read as "If the address of p is a, then p is a people and a is an address." The square brackets in the above declaration indicate the functional nature of the relationship between a people and his/her address.

Similar to how we add the facts to entities, we can assert facts for functional predicates. For example, if Jason's address is "XYZ123", then we can indicate that fact as follows:

```
+addressOf["Jason"] = "XYZ123".
```

The address fact "XYZ123" has to be in the *Address* entity:

```
+Address(a), +hasAddressName(a:"XYZ123").
```

We can also use a unary form of *functional predicate* to represent same concept:

```
isOfAddress(p, a) -> People(p), Address(a).
```

The above statement can be read as "If p is of address a, then p is a people and a is an address".

Some facts are simply asserted to be true, using the delta modifiers, while other facts are computed by applying a *derivation rules* to the facts that are already known. For example, if we know the length and width of a rectangle, we can calculate the area of it by multiplying its length by its width.

For example, if we want to express the relationship like parenthood and sibling relationship among different people, we first can define a predicate:

```
parentOf[p1] = p2 -> People(p1), People(p2).
```

Now consider a rule to derive the sibling relationship. Two different people are siblings of each other if they share a parent. We define a unary predicate isSiblingOf:

```
isSiblingOf(p1, p2) -> People(p1), People(p2).
```

With *parentOf* and *isSiblingOf* we can define our first *derivation rule*:

```
isSiblingOf(p1, p2) <- parentOf[p1] = p3, parentOf[p2] = p3, p1 ! = p2.
```

We can read this rule as ”p1 is a sibling of p2 if there is some p3 such that p1 and p2 have p3 as a parent, and p1 is not the same as p2.”

The inequality is expressed using the not-equals operator ”!=”, and comma is still means and operator. Note that the direction of the arrow in rules is from right to left. We can see that an arrow directed to the right is used in constraints, and if the formula that comes after the arrow is true, then the formula before the arrow will also be true.

With the basic knowledge of *entity*, *functional predicate*, and *declaring derivation rules* we can infer some facts based on what we have already known.

For example, consider we have *People* entity, *parentOf* functional predicate, and *isSiblingOf* derivation rule, and we add following facts to the database system:

```
+People(p), +hasPeopleName(p:"Allen").
```

```
+People(p), +hasPeopleName(p:"Bill").
```

```
+People(p), +hasPeopleName(p:"Clare").
```

```
+parentOf["Allen"] = "Bill".
```

```
+parentOf["Clare"] = "Bill".
```

Based on the above information, and the *isSiblingOf* derivation rule:

```
isSiblingOf(p1, p2) <- parentOf[p1] = p3, parentOf[p2] = p3, p1 ! = p2.
```

The system will automatically infers the fact

```
isSiblingOf("Allen", "Clare")
```

In this research, we encode the type constraint system into a set of *entity*, *functional predicate*, and *declaring derivation rules*. And according to the facts we have already known, the system would automatically infer the rest of facts for us. The details of the *LogiQL* encoding is explained in Section 3.3.

2.4 Related Work

Checker Framework [7, 27] includes many pluggable type systems that express and enforce compile-time properties. Nullness type system for example is a type system that prevent null pointer exception in compile time. The corresponding checker checks possibly dereference of null expression, iteration over possibly-null collections in enhanced for loop, and accessing a possibly-null array. Another example is Interning checker, which checks the reference quantity tests to prevent the misuse of the equal operator "==".

Generic Universe Types [17] is another type system implemented on the top of Checker Framework. It's an ownership type system that organizes the heap hierarchically and groups objects by owner. The ownership encapsulation can make sure that the modifications for object wouldn't across the owner's boundaries. Programmer can annotate the program with GUT type qualifiers, and the program will need to obey the ownership rules described by the qualifiers.

Outside of Checker Framework, there are other type systems have been proposed. Flanagan and Freund [20] for example presented a type checker called *rccjava*, which contains a type system that ensures that a Java program doesn't contain any race conditions. Boyapati and Rinard [16] took advantage of *rccjava* and made modifications such that programmers can write generic code to implement a class, then the different objects of the same class can have different protection mechanisms. Programmers can specify the mechanism for the object when then define the type of the variables. The resulting language is called Parameterized Race Free Java.

Besides Checker Framework, there are some other tools that can also allows users to add user-defined type qualifiers to Java. JavaCOP [14] is such tool that defines a declarative, rule-based language that can express rules as constraints on AST nodes. It can process the information from both annotated additional types and Java's own type system. JQual [22] can adds pluggable type qualifiers to Java, and also performs type inference for pluggable user-defined type qualifiers. The tool traverses the method bodies in the program and generates subtyping constraints, and then resolve the constraints. Ekman and Hedin [19] presents an extensible Java Complier including a component that can check and infer non-null types through a non-nil type system.

There are various of type inference approaches has been proposed for pluggable type systems.

Tunable Static Inference for Generic Universe Types [18] presents a type inference approach for Generic Universe Types, and it also provides the theory foundation for Checker

Framework Inference and this work. The paper introduces a constraint-based type inference approach, which is also implemented by *Checker Framework Inference*. It proposes a way to encode the constraints of Generic Universe Types as a Boolean satisfiability problem, which inspires the Max-SAT encoding in Section 3.2.1. The main difference to our Max-SAT encoding is that the Max-SAT encoding in [18] is only for GUT type system, but in our work, we generalized the approach, and make it usable for any pluggable type systems.

For *rccjava*, Flanagan and Freund later proposed a constraint-based type inference approach [21]. In their work, constraints system are reduced to SAT problem, and they use SAT solver to find the solutions. But the solver may provide any valid inference solution, and sometimes it makes the solution trivial.

Besides constraint-based type inference, Agarwal and Stoller [11] presents a run-time approach to type inference for Parameterized Race Free Java. The approach monitors some executions of the program and infer possible types according to the observed behavior. However the downside of run-time inference is that the result may be unsound, since the observed behavior may not express all possible behaviors of the program.

The Constraint Graph Generator described in Section 3.1.2 converts type constraints into a graph representation, and performs graph traversal algorithms to separate the graph. Kodumal and Aiken [24] described a way that uses graph reachability problem to solve set constraint problem. The approach treats the constraint system as a directed graph. The nodes of the graph are set expressions, and the edges are constraints. The edge could either be successor edge or predecessor edge. Then the constraint system can be solved by applying the algorithm described in [12]. Kodumal and Aiken's approach is overqualified for our problem, since in our situation we would like to either identify independent components in constraint system or find out the all reachable nodes from certain a node, and both of them can be simply achieved through Breadth-first Search algorithm.

In this thesis, we built a *Dataflow Type System* to infer all possible Java types of method and variable. Soot, an analysis and optimization framework for Java, provides SPARK framework [26] that can perform similar analysis. SPARK is a points-to analysis tool that can perform equality- and subset-based analyses, and return a set of possible targets for given variable. Soot takes Java bytecode as input, and converts it to intermediate representations for example, JimpleBody, ShimpleBody, etc. Then it applies some analyses and optimization, and converts the result back to bytecode. SPARK uses the JimpleBody intermediate representations as the input, generates pointer assignment graph, and then calculate the points-to set for a veriable according to the graph. We can retrieve all possible types from the points-to set.

WALA is another static analysis tool that can perform points-to analysis [28]. The tool implements an Andersen's-style analysis [13] by constructing a flow graph representing the pointer flow for a program and computes a points-to set for each variable. In points-to set, we can get all possible types for given variable.

In Checker Framework, it provides a *Constant Value Checker* [4], which can determine whether variables' value can be known at compile time. The checker can infer the type qualifiers for variables but not for fields and method signatures, and like *Dataflow Type System*, the type qualifiers in Constant Value Checker can take an argument as a set of values. The checker can determine values with type boolean, double, integer and string. The difference between *Constant Value Checker* and *Dataflow Type System* is the checker can determine both type and value, and Dataflow Type System only focuses on type. However, besides the four types that the checker can infer, Dataflow Type System can infer all primitive types, reference types, and array types, and it also can infer the type qualifiers for fields and method signatures.

Chapter 3

Type Constraint Solver

This chapter discusses a system called *Type Constraint Solver* that solves the type constraints generated by *Checker Framework Inference*, such that we can get the concrete type qualifier for each slot. *Type Constraint Solver* independents to specific type systems, which means it has a general constraint solving mechanism that deal with the constraints generated from arbitrary type systems. Some type systems, for example *Dataflow Type System* in Chapter 4, has it's own requirement for constraint solving. *Type Constraint Solver* can also be easily extended with custom solvers. The system provides two kinds of encoding strategy that can solve the Section 3.1 presents the whole structure of the solver. The solver consists of a *front end*, *constraint graph generator*, *serializers*, and different *back ends* performing constraints solving. Section 3.2 shows one of the back ends: *Max-SAT back end*. It encodes all types of constraint as a weighted SAT problem, solves the problem by calling existing SAT solver, and decodes the output from SAT solver to a set of concrete result for the program. Section 3.3 describes another back end: *logiQL back end*. The section explains how to reduce the type constraints to statements of logiQL language, and transform the result from LogicBlox.

3.1 Type Constraint Solver Structure

The system is designed for solving constraints from arbitrary type system through different encoding strategies. Type Constraint Solver provides multiple back ends that can solve the constraints, and each back end corresponds to one kind of encoding for the constraints. The structure also supports adding new back ends easily. Figure 3.1 overviews the whole structure.

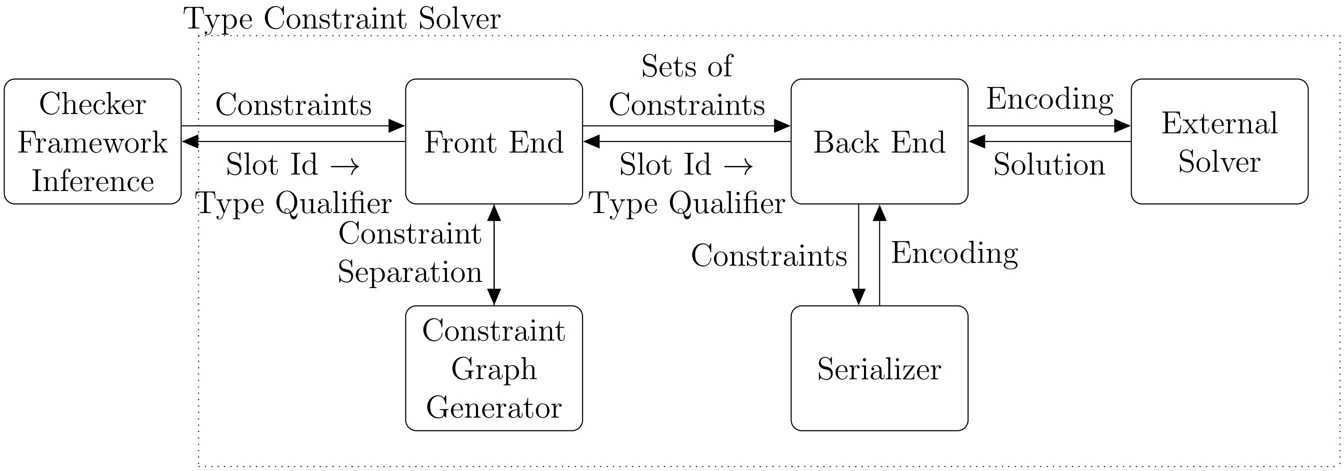

Figure 3.1: Overview of Type Constraint Solver Structure

Front end takes the output (type constraint) from *Checker Framework Inference* and performs some initialization steps. Then we have an option to use *Constraint Graph Generator* to separate constraints into different groups, which could be solved separately. After the separation, *Front End* would send sets of constraint to one type of the back ends. *Back end* would invoke *Serializer* first. *Serializer* is the tool that can convert different types of constraint into another form which can be solved by external solvers. The encoding process happens in *Serializer*. Then *Back end* takes the encoding from *Serializer* and sends it to an external solver. After external solver gives the solution back to *Back end*, it decodes the solution and passes the decoded solution to *Front End*. *Front End* then returns the solution to *Checker Framework Inference*. Sections 3.1.1 to 3.1.4 explain every components in Figure 3.1 in detail.

In the whole process, *Constraint Graph Generator* is an optional tool to choose. So if we don't separate the constraint into different sets, *Front End* will pass all of the constraints to *Back End* directly. The details of *Constraint Graph Generator* will be discussed in Section 3.1.2. The solution from external solver will be decoded by *Back End*, and the decoded form is a map between slot Id and the type qualifier that would be annotated to this slot. Currently, we have two back ends can use, *Max-SAT* back end and *LogiQL* back end. Figure 3.2 shows a more detailed structure of *Type Constraint Solver* with the two back ends. *Max-SAT* serializer converts constraints to CNF formulas, and the formulas are solved by existing Max-SAT solvers. *LogiQL* serializer converts constraints to LogiQL language, and LogicBlox can process the language and generate the result. Section 3.2 and 3.3 will discuss Max-SAT back end and LogiQL back end in detail.

Note that the relationship between back end and serializer is simplified in Figure 3.2. The back end doesn't directly connect to the corresponding serializer, instead, there is a deliverer between the back end and serializer. The details about the deliverer will be discussed in Section 3.1.3.

3.1.1 Front End

The front end implements the interface *InferenceSolver* in *Checker Framework Inference*. As we introduced in Section 2.1, *Checker Framework Inference* provides constraints, slots, command line configurations, and all other necessary information to solver through this interface.

The motivation of the front end is that we want a way that can control different back ends easily. Before the front end is adapted, different back ends and serializers are developed independently. There are some disadvantages of independent development: although the main implementation of different back ends are different, there are still some common logic shared among each other. So developing them independently would produce some duplicate and unreusable code for example, some configurations only need to be set once, instead of multiple times in different back ends. Moreover, we wanted new back ends can be easily plugged in to the system, so front end provides all the necessary APIs to be extended and overridden such that the developers don't need to build everything from scratch.

The main responsibilities of front end are recognizing the command line arguments provided by users, making a series of configurations, creating the right serializer and back end that are indicated by user, invoking back end, and returning the output of back end to *Checker Framework Inference*. At the moment, the possible command line arguments are: the kind of back end and serializer, whether use constraint graph to separate the constraints, and if the constraints are separated, solve them in parallel or sequential.

For the creation of back end, we took advantage of Java Reflection API. Users indicate the type of back end in command line, and the corresponding back end is created in run time reflectively. Instead of hard coding the creation of different back ends, reflective creation gives the user flexibility to choose different back end. Constraints and Serializer are two necessary components for back end creation. Constraints are generated by *Checker Framework Inference*, and serializer is also reflectively created. Note that at the moment, each back end only corresponds to one serializer, so the back end type indicated by user in command line is also the type of the serializer. More details about serializer will be

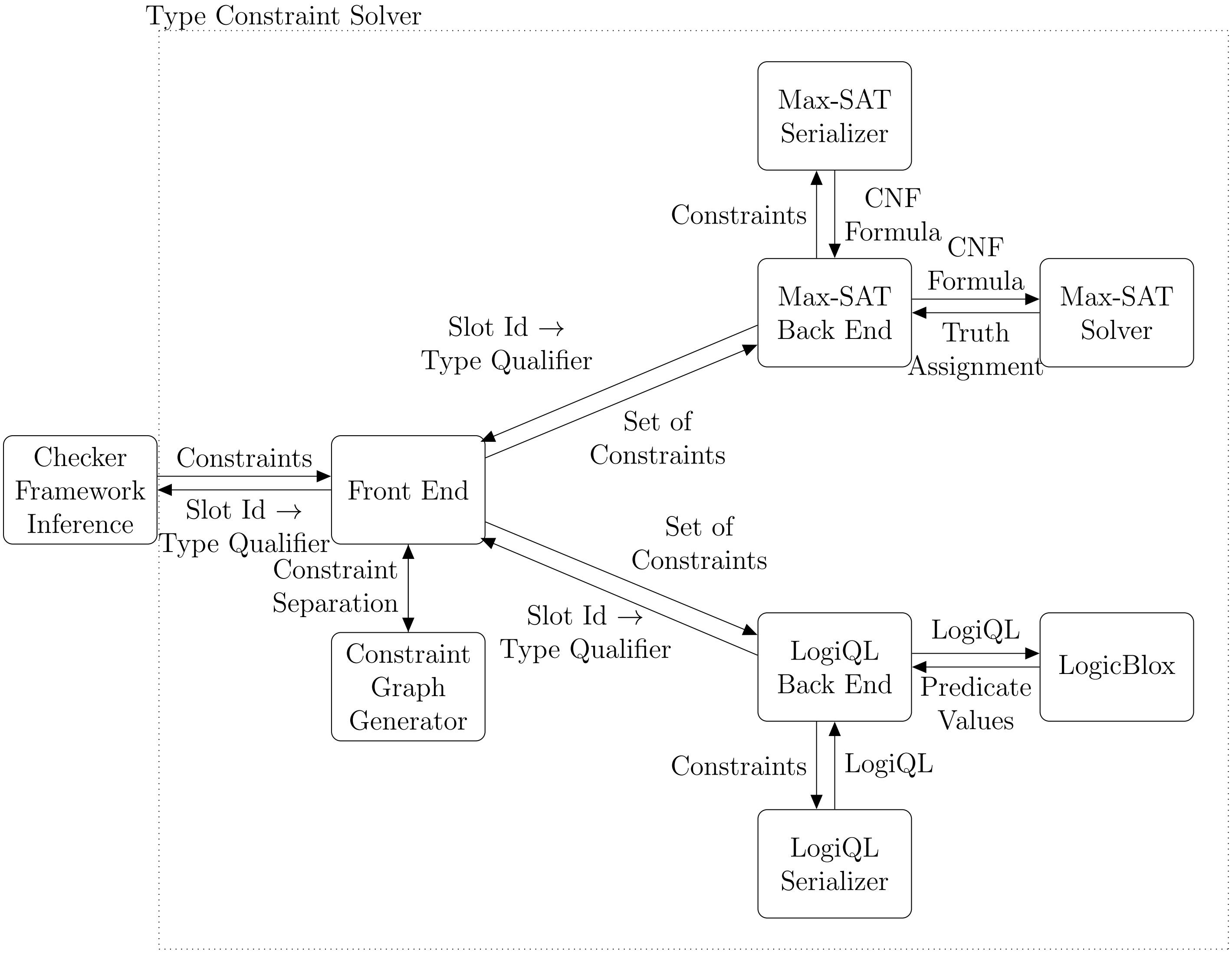

Figure 3.2: Detailed Structure of Type Constraint Solver

discussed in Section 3.1.3. If users don't provide any arguments, then the default configuration will be set: *Max-SAT* back end and serializer, perform the constraint separation, and solving the sets of constraint in parallel.

Constraint Graph is a graph representation for constraints. Constraints can be separated into different sets through graph manipulation, and then we can solve them independently. After this separation, we have options to solve the sets of constraints in parallel or sequential. For both cases, we create a list of back ends with different sets of constraint. If we decide to solve them in sequential, then each back end will be invoked one by one. Sets of constraint can also be solved in parallel: we used a thread pool and put back ends in different threads through a Callable object, so the back ends can solve the constraints at same time. We took advantage of Future class in Java, such that we can get the calculated result from each each back from a list of Future object. After the solving processes finished, the results from different back ends are going to be merged. The details of Constraint Graph will be discussed in Section 3.1.2.

Once the back end finished solving the constraints, it returns a result, which is a map from slot Ids to the type qualifier for that id, to front end. Then front end can print the result out on screen or write it into a file, and return it to *Checker Framework Inference*.

3.1.2 Constraint Graph

After *Checker Framework Inference* generates a set of type constraint, instead of sending constraints to a back end directly, we have an option to separate the constraints into different groups. The separation option is designed for two reasons: first, for some special type systems like *Dataflow* system, which will be introduced in later section in detail, this separation analysis is necessary. Second, solving different groups of constraints separately is good for performance, for example, the time complexity for *SAT* solver is exponential, so it's good for performance if we can divide the input for *SAT* solver into different independent groups and solve them separately.

We convert the constraints to a directed graph, and we simply call it *Constraint Graph*. Each slot in constraints will be represented by a vertex in the graph, and each constraint will be represented by one or more edges in the graph. The relationship among slots and constraints can be represented by the relationship among these vertices and edges.

We use a graph representation for two reasons. First, in the original constraint representation, there is no way to know the reachability among all of the slots. Type qualifier for one slot could influence others if these slots are connected by some constraints, so we want to know the reachable slots from the given slot. In graphs, one node is reachable

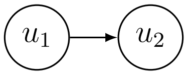

Figure 3.3: Constraint Graph for a Single Subtype Constraint $u_1 <: u_2$

from another, if there is a path between them. Second, we can perform lots of graph related algorithms easily if we convert the constraints to a graph. Currently we have two constraint separation strategies, which will be discussed later, and they are all based on breadth-first search.

We use instances of Java class *Edge* and *Vertex* to construct the graph. Each edge has the reference to the "from" and "to" vertexes and the corresponding constraint. Each vertex has the reference to all edges that are incident to it and the corresponding slot. A vertex is a constant vertex if it refers to a constant slot. For subtype constraint, the direction of the edge is from subtype slot to supertype slot. Figure 3.3 shows the graph for one subtype constraint: $u_1 <: u_2$. For adaptation constraint, there are three nodes in total, so we connect them together by four edges. Figure 3.4 shows the graph for one adaptation constraint: $u_1 \triangleright u_2 = \alpha_3$. For all other kinds of constraints, the constraint and two slots can consist two nodes and two edges in both directions.

There are two separation strategies:

Strategy one: separating the constraint graph by constant. We start at each constant vertex, run breadth-first search, and collect all the edges that can be found. This separation strategy can give us the information that which constrains can be reached from each constant slot.

Strategy two: finding all of the weakly connected components, and put the edges in different components into different groups. "Weakly connected" here means replacing all of directed edges with undirected edges produces a connected (undirected) graph[3]. Every slots and constraints in one component have no relation with the slots or constraints outside of the component, so the solution for one component is independent to others.

Figure 3.5 shows a complete constraint graph. Label c and v stands for constant slot and variable slot separately in the graph.

After constructing the graph, we traverse it and group the edges by different separation strategies.

For strategy one, in Figure 3.5, there are two constant vertices c_1 and c_2. Starting from c_1, we can find v_4, v_5 and the edges among those vertices. Starting from c_2, we can find v_9,

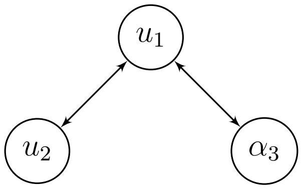

Figure 3.4: Constraint Graph for a Single Adaptation Constraint $u_1 \triangleright u_2 = \alpha_3$

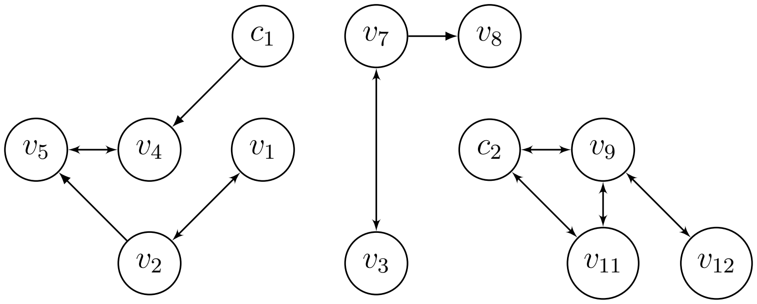

Figure 3.5: An Example of Constraint Graph

v_{11}, v_{12}, and the edges incident to them. The algorithm for finding these paths is based on *breadth-first search*, and it listed in Algorithm 1.

After running the algorithm, we map every constant vertex to a set of edge.

For strategy two, in Figure 3.5, there are three weakly connected components. And the algorithm for finding these components is Algorithm 2

If there are total n constraints, m slots, and k constant slots. The time complexity of Algorithm 1 is $\mathcal{O}\big(k \times (m+n)\big)$, and the time complexity of Algorithm 2 is $\mathcal{O}(m+n)$. During the constraint graph construction, the system iterates all constraints and creates corresponding edge and vertices for each constraint. Since the number of slots in each constraint is no larger than 3, the time complexity of graph construction is $\mathcal{O}(n)$. So the time complexity of whole constraint graph generator is $\mathcal{O}\big(k \times (m+n)\big)$.

One thing need to be mentioned here is that, strategy one can only be used for *Dataflow* type system at the moment, and after solving the sets of constraint separately, the way that we merge the solutions together is special and contains *Dataflow Type System* specific implementation, which will be discussed in *Dataflow* Chapter. However, for strategy two, method for merging different solutions are straight forward: just simply combining different

Algorithm 1 Algorithm for separating the constraint graph by constant

```
let independentPath be a map between constant vertex and a constraint set.
for Every constant vertex v in constraint graph g do
    let independentConstraints be an empty set
    let Q be an empty queue
    let visited be an empty set
    Q.enqueue(v)
    while Q is not empty do
        current = Q.dequeue()
        visited.add(current)
        for each vertex u that is adjacent to current,
        and edge e between u and current do
            independentConstraints.add(e)
            if u is not in visited then
                Q.enqueue(u)
            end if
        end for
    end while
    independentPath.put(v, independentConstraints)
end for
```

solutions together. We can do this combination because different sets of constraint are completely independent, so the solutions for different sets doesn't affect each other.

Algorithm 2 Algorithm for finding weakly connected components

```
let visited be an empty set
let result be an empty list
for Every vertex v in constraint graph g do
    if v is not in visited then
        let independentPath be an empty set
        let $Q$ be an empty queue
        $Q$.enqueue(v)
        while $Q$ is not empty do
            $current = Q$.dequeue()
            visited.add($current$)
            for each vertex u that is adjacent to current,
            and edge e between u and current do
                independentPath.add(e)
                if u is not in visited then
                    $Q$.enqueue(u)
                end if
            end for
        end while
        result.add(independentPath)
    end if
end for
```

For the components that don't contain any constant slot, how we would annotate the variable slots depends on the type system. For Dataflow type system, every slot in such component should not be annotated any type qualifier, so we just ignore it. For some other type systems, we can annotate slots with the default type qualifiers.

In summary, Algorithm 1 can only be used in the inference for Dataflow type system for now, and it's a necessary step for the inference. For Algorithm 2, it's a general constraint separation strategy, which can be used for systems without any special requirements, like OsTrusted type system. However, for a new type system that is defined on Checker Framework Inference, neither of Algorithm 1 nor Algorithm 2 may be suitable for it. In that case, if constraint separation is mandatory, a new constraint separation approach needs to be come up with.

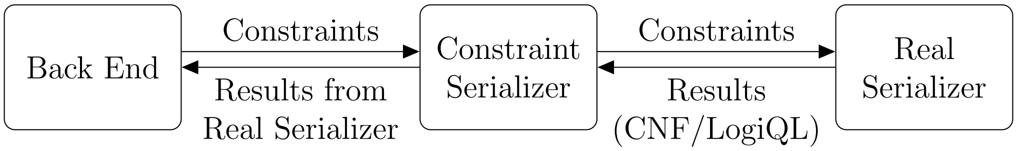

Figure 3.6: Relationship among Back End, ConstraintSerializer, and Real Serializer

3.1.3 Serializer

Serializer is the tool that encodes constraint into other forms for example, Max-SAT problem or statements of LogiQL language. It can automatically transform constraints into corresponding form of encodings, either statements of LogiQL language or Boolean formulas, and external solvers are able to solve them. The two current encodings, *Max-SAT* encoding and *LogiQL* encoding, calculate the encoding according to the type of constraints, slot Ids in constraints, and qualifier hierarchy in the underlying type system. In *Max-SAT* encoding, for each slot Id and type qualifier, we generate a unique integer value for that Id and qualifier. In *LogiQL* encoding, we have an entity to store all slot Ids. For each type qualifier, we used the qualifier's as a part of the predicate's name, and use declaring derivation rules to connect and calculate values in different predicates.

The details of *Max-SAT* and *LogiQL* encoding will be discussed in Section 3.2.1 and Section 3.3.1.

Encoding constraints is the core part of constraint solving. Constraints themselves defines judgments over class, field, and method declarations as well as over expressions[18], but it's not very straightforwards to express the exact type qualifier for each location. So we reduce the constraint system to other problems, and by solving those problems and decoding the results, we can get the type qualifier for each location with satisfaction of constraint.

As we mentioned in the beginning of this chapter, there is a deliverer between back end and serializer, in the real implementation, we call it ConstraintSerializer. ConstraintSerializer warps up the real serializer, and for each kind of constraint, instead of serializing it, ConstraintSerializer delivers the constraint to the real serializer and returns what the real serializer returns. The relationship among back end, ConstraintSerializer, and real serializer is showed in Figure 3.6.

ConstraintSerializer is designed for the scalability of the whole system. For most types of constraint, the logic for serializing them is fixed and already has been implemented in real serializers because the logic doesn't depend on the type system. However sometimes it cannot handle all of the situations without any type system related implementation. Adaptation constraint is special: the encoding for adaptation constraint depends on the underlying type system. So if someone wants to develop a constraint solver for the type system with viewpoint adaptation like Generic Universe Type System[18], he or she needs to implement the Generic Universe Type System specific encoding, and incorporating with the ConstraintSerializer would make the implementation easier. The developers only need to make a subclass of ConstraintSerializer, and override the serialize method for adaptation constraint with the type system specific logic. For other kinds of constraint, they will be serialized as usual.

In Section 3.1.1, we mentioned that real serializer is reflectively created according to the command line arguments provided by users. Once it receives the constraint from the deliverer, serializer will serialize the constraint according to its kind.

3.1.4 Back End

Back end calls serializer converting constraints into other forms, once conversion is finished, back end uses corresponding solver to solve those forms, decode solutions from the solver, and return the decoded result to front end. Figure 3.7 shows the different phases in back end.

Back end is meant to be fixed and non-extendable in terms of usage, which means if people want to use the system, they don't need to change any lines of code in back end, and don't need to make any subclass of back end because the system is designed for dealing with constraints from arbitrary type systems. (Although we may change the implementation of back ends in order to enhance it's functionality in the future) As we discussed in Section 3.1.3, when we deal with some complicated type systems like Generic Universe Type System[18], we only need to extend ConstraintSerializer and put the type system specific logic in there without touching anything in back end.

Currently the system supports two back ends: Max-SAT back end and LogiQL back end. Constraints are encoded as CNF formulas and statements of LogiQL language in Max-SAT back end and LogiQL back end respectively. The details of encoding will be introduced in Section 3.2.1 and Section 3.3.1. For the solver, the middle part in Figure 3.7, it might be a separate non-Java project. For LogiQL back end, the solver is LogicBlox which is a logic database running with LogiQL language. And for Max-SAT back end, the solver is existing SAT solvers. We use two SAT solvers: *sat4j*[15] and *lingeling* [5].

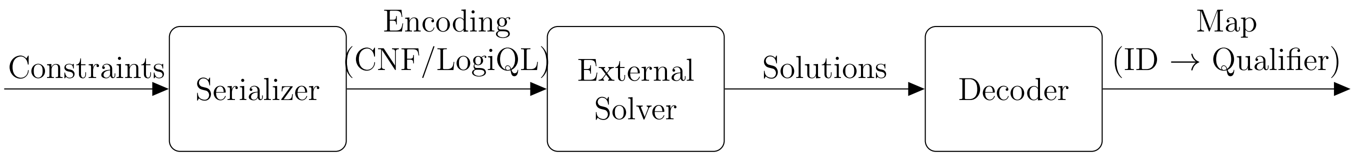

Figure 3.7: Phases in Back End

Section 3.3.2 and Section 3.2.2 discuss the solvers that we are using. The implementation of decoder depends on the encoding, and it will be explained in Section 3.2.3 and Section 3.3.3.

Whether the solutions given by different back ends are identical to each other depends on the underlying type system. For example, for OsTrusted type system discussed in Section 2.1.2, if a variable slot is subtype of OsUnTrusted type, then the type qualifier for that slot could be either OsUnTrusted or OsTrusted. In this case, as long as the solutions from different back ends satisfy the type constraint, then the solutions may not be unique. However, for Dataflow type system in Chapter 4, even if there could be multiple solutions, since the possible run-time types of each location are set once the program had been written, there should always be a unique best solution. See Chapter 4 for more details.

3.2 Max-SAT Back End

This section discusses all components in Max-SAT back end. Section 3.2.1 shows how to encode all kinds of constraint as a SAT problem. Section 3.2.2 discusses the SAT solvers that are used in the back end. Section 3.2.3 describes how to transform the output of SAT solver to the type qualifier for each slot in the program.

3.2.1 Max-SAT Encoding

This section explains how to encode the constraints as Boolean formulas. This conversion is done by Max-SAT serializer.

Max-SAT is the abbreviation of *maximum satisfiability problem*, which is the problem of determining the maximum number of clauses, of a given Boolean formula in conjunctive normal form, that can be made true by an assignment of truth values to the variables of the formula.

The reasons that we encoded constraints to *Max-SAT* are: first, for each constraint variable and one type qualifier, the qualifier could be either annotated for that variable or not, and this is easily to be expressed as a Boolean value. Second, for some type systems, like Generic Universe Type System, the weights are attached to some constraints, so in that type system, each constraint variable, the type system uses the position of variable in AST to encode a preference[18]. And *SAT* solver allows us to add weights to some clauses that would make the encoding much easier.

For any type system, the *Max-SAT* encoding for *Subtype Constraint*, *Equality Constraint*, *Inequality Constraint*, *Comparable Constraint*, and *Preference Constraint* can be generated automatically according to the type hierarchy, but for *Adaptation Constraint*, the encoding for it can only be generated manually because the viewpoint adaptation rules depend on the type rules of the underling type system. As we mentioned before, the encoding for *Adaptation Constraint* is easy to be added. What the user need to do is overriding the *serialize* method for *Adaptation Constraint* in *ConstraintSerializer*, and adding the right behaviors according to the typing rules. The following part explains automatic encoding for other constraints.

The first step of the encoding is we map every slot id to some integer values, and an integer value will be a Boolean literal in CNF formula. For every type qualifier in the pluggable type system, we use a constant integer value to represent it. The constant integer value starts from 0 to the number of type qualifier k in the underlying type system. If we consider type system T, the hierarchy of T is shown in Figure 3.8, there are four type qualifiers τ_1, τ_2, τ_3, τ_4, then we use 0 to represent τ_1, 1 for τ_2, 2 for τ_3, and 3 for τ_4. The order for different qualifiers doesn't matter, and the integer values are randomly assigned. Algorithm 3 shows that how this mapping procedure works. With these constant integer, the mapping rule is: we add the constant value to the number of type qualifier in underlying type system times the difference of slot's id minus 1, and the result represents the annotated status of the slot with the corresponding type qualifier, and we use that integer in Max-SAT formula. Algorithm 4 *GETSATID* shows how to perform the above calculation. So if there are m type qualifiers in the type system, for any slot id n, the integer values are from

$$(n-1) \times m$$

to

$$(n-1) \times m + (m-1)$$

They represent all the type qualifiers for that slot. Figure 3.9 shows the mapping table for type system T.

Algorithm 3 Map from type qualifiers to integer values

```
Function: MAP_QUAL_INT($\bar{t}$)
Input: All type qualifiers in underling type system $t$.
Output: A map that maps from each type qualifier to an integer value.
Let integer $i$ with initial value 0
Let map be an empty map
for each type qualifier $t$ in $\bar{t}$ do
    Put $(t, i)$ in map
    $i \leftarrow i + 1$
end for
return map
```

Algorithm 4 Map from Slot Id and type qualifier to an integer Id in Max-SAT formula

```
Function: GETSATID($n$, $m$, $t$)
Input: Slot Id $n$, number of type qualifiers in underlying type system $m$, type qualifier $t$.
Output: Integer Id in Max-SAT formula for $n$ and $t$.
For type qualifier $t$, get it's corresponding integer representation $k$ from the map generated by Algorithm 3.
return $(n - 1) \times m + k$
```

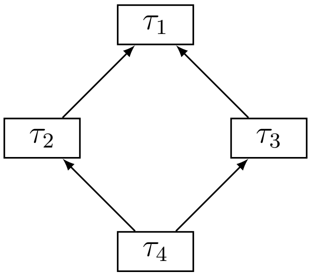

Figure 3.8: Type Hierarchy of Type System T

slot id \ qualifiers	τ_1	τ_2	τ_3	τ_4
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11
...	...	...	...	...
n	$(n-1) \times 4$	$(n-1) \times 4 + 1$	$(n-1) \times 4 + 2$	$(n-1) \times 4 + 3$

Figure 3.9: Mapping Table (from slot Id to integer values) for Type System T

The second step is going through all the type constraints, and generating the Boolean formula encoding for each one. The approach for encoding different constraints are listed below (The type qualifiers used in following content are all from type system T):

- *Encoding for Equality Constraint*

  For given equality constraint, if the constraint is between a variable slot v and a constant slot τ, $v = \tau$, then it means the annotation of the variable slot v has to be τ. We will first look at the mapping table, and calculate the corresponding mapping integer of v and τ. We use β_v^τ to represent the integer value. Then, we use a single clause to encode equality constraint $v = \tau$:

  $$(\beta_v^\tau)$$

  For example, in type system T, if the constraint is $n = \tau_1$, the encoded clause would be:

$$GETSATID(n, 4, \tau_1)$$

If the equality constraint is between two variable slots v_1 and v_2, then we will use logic

$$(\neg v_1 \ \vee \ v_2) \ \wedge \ (v_1 \ \vee \ \neg v_2)$$

to express that v_1 equals to v_2. We need to go through each type qualifier τ_i one by one in type system, and generate the clauses for each of them. The general form of the clauses is

$$(\neg \beta_{v_1}^{\tau_i} \vee \beta_{v_2}^{\tau_i}) \wedge (\beta_{v_1}^{\tau_i} \vee \neg \beta_{v_2}^{\tau_i})$$

For example, If the Equality Constraint between n_1 and n_2 is generated based on type system T in Figure 3.9, then the encoding would be:

$$\begin{aligned}
&\big(\neg GETSATID(n_1, 4, \tau_1) \vee GETSATID(n_2, 4, \tau_1)\big)\\
\wedge&\big(\neg GETSATID(n_2, 4, \tau_1) \vee GETSATID(n_1, 4, \tau_1)\big)\\
\wedge&\big(\neg GETSATID(n_1, 4, \tau_2) \vee GETSATID(n_2, 4, \tau_2)\big)\\
\wedge&\big(\neg GETSATID(n_2, 4, \tau_2) \vee GETSATID(n_1, 4, \tau_2)\big)\\
\wedge&\big(\neg GETSATID(n_1, 4, \tau_3) \vee GETSATID(n_2, 4, \tau_3)\big)\\
\wedge&\big(\neg GETSATID(n_2, 4, \tau_3) \vee GETSATID(n_1, 4, \tau_3)\big)\\
\wedge&\big(\neg GETSATID(n_1, 4, \tau_4) \vee GETSATID(n_2, 4, \tau_4)\big)\\
\wedge&\big(\neg GETSATID(n_2, 4, \tau_4) \vee GETSATID(n_1, 4, \tau_4)\big)
\end{aligned}$$

- *Encoding for Inequality Constraint*

  For inequality constraint, if the constraint is between a variable v and constant τ, similar to equality constraint, we use a single integer as a clause, but we need to use negated integer to express the type of v must not be τ:

  $$(\neg \beta_v^\tau)$$

In type system T, the encoding for inequality constraint $n \neq \tau_1$ is

$$\neg GETSATID(n, 4, \tau_1)$$

If the inequality constraint is between two variable slots v_1 and v_2, we can use clauses

$$(\neg v_1 \;\vee\; \neg v_2) \;\wedge\; (v_1 \;\vee\; v_2)$$

to express that v_1 doesn't equal to v_2. Similar to equality constraint, we will need to go through each of the type qualifier τ_i, and generate encoding for all of them in the form:

$$(\neg \beta_{v_1}^{\tau_i} \vee \neg \beta_{v_2}^{\tau_i}) \wedge (\beta_{v_1}^{\tau_i} \vee \beta_{v_2}^{\tau_i})$$

For example, the encoding for an inequality constraint between n_1 and n_2 in type system T would be:

$$\begin{aligned}
&\big(\neg GETSATID(n_1, 4, \tau_1) \vee \neg GETSATID(n_2, 4, \tau_1)\big) \\
&\wedge\big(GETSATID(n_2, 4, \tau_1) \vee GETSATID(n_1, 4, \tau_1)\big) \\
&\wedge\big(\neg GETSATID(n_1, 4, \tau_2) \vee \neg GETSATID(n_2, 4, \tau_2)\big) \\
&\wedge\big(GETSATID(n_2, 4, \tau_2) \vee GETSATID(n_1, 4, \tau_2)\big) \\
&\wedge\big(\neg GETSATID(n_1, 4, \tau_3) \vee \neg GETSATID(n_2, 4, \tau_3)\big) \\
&\wedge\big(GETSATID(n_2, 4, \tau_3) \vee GETSATID(n_1, 4, \tau_3)\big) \\
&\wedge\big(\neg GETSATID(n_1, 4, \tau_4) \vee \neg GETSATID(n_2, 4, \tau_4)\big) \\
&\wedge\big(GETSATID(n_2, 4, \tau_4) \vee GETSATID(n_1, 4, \tau_4)\big)
\end{aligned}$$

- *Encoding for Comparable Constraint*

  Encoding for comparable constraint is very similar to inequality constraint, but the difference is if v_1 is comparable to v_2, then v_2 doesn't equal to the types that v_1 is incomparable with.

  If we have comparable constraint ($v <:> \tau$), a variable slot v between a constant τ, we will encode the constraint as:

$$(\neg \beta_v^{\kappa})$$

In the underlying type system, κ and τ are incomparable to each other.

If the comparable constraint is between two variable slots v_1 and v_2, (v_1 <:> v_2), we will use the logic

$$(\neg v_1 \ \vee \ \neg v_3) \ \wedge \ (v_1 \ \vee \ v_3)$$

to express v_1 is not equal to v_3. In the underlying type system, v_2 is incomparable to v_3.

We still need to go through each type qualifier τ_i in the underlying type system, find the all incomparable type qualifier τ_j of τ_i, and use the following form as the encoded clauses.

$$(\neg\beta_{v_1}^{\tau_i} \vee \neg\beta_{v_2}^{\tau_j}) \wedge (\beta_{v_1}^{\tau_i} \vee \beta_{v_2}^{\tau_j})$$

In type system T, τ_1 is direct supertype of τ_2 and τ_3, τ_4 is direct subtype of τ_2 and τ_3. τ_2 and τ_3 are incomparable to each other, so if the constraint is between a variable n and constant τ_2, we treat it same as the inequality constraint between n and τ_3:

$$\neg GETSATID(n, 4, \tau_3)$$

If the constraint is between two variable slots n_1 and n_2, the encoding would be:

$$\begin{aligned} &\big(\neg GETSATID(n_1, 4, \tau_2) \vee \neg GETSATID(n_2, 4, \tau_3)\big) \\ &\wedge\big(GETSATID(n_2, 4, \tau_3) \vee GETSATID(n_1, 4, \tau_2)\big) \\ &\wedge\big(\neg GETSATID(n_1, 4, \tau_3) \vee \neg GETSATID(n_2, 4, \tau_2)\big) \\ &\wedge\big(GETSATID(n_2, 4, \tau_2) \vee GETSATID(n_1, 4, \tau_3)\big) \end{aligned}$$

Above clauses indicate the facts that n_1 annotated by τ_2 and n_2 annotated by τ_3, and n_1 annotated by τ_3 and n_2 annotated by τ_2 cannot be true.

- *Encoding for Subtype Constraint*

  Encoding for subtype constraint is more complicated than others, there are following situations:

  - Case one (v <: τ):

    If a variable v is subtype of a constant τ, v <: τ, then we first check whether τ is the bottom type in the underlying type system, if it is, then v has to be

τ. And we can use the exact same logic in equality constraint to encode the subtype constraint:

$$(\beta_v^\tau)$$

In type system T, τ_4 is the bottom type, so the encoding of subtype constraint $(n <: \tau_4)$ would be:

$$GETSATID(n, 4, \tau_4)$$

If τ is not the bottom type, then v cannot be strict supertype of τ. Then for each τ's strict supertype ρ, we generate a single clause

$$(\neg\beta_v^\rho)$$

to express the subtype constraint.

For example, for subtype constraint $n <: \tau$, if τ is τ_2 in type T, the encoding would be:

$$\neg GETSATID(n, 4, \tau_1)$$

- Case two $(\tau <: v)$:

  If a constant τ is subtype of a variable v, $\tau <: v$, then we still need to check the position of τ in hierarchy of the underlying type system. If it's the top type, then then v has to be τ. In this case, we will encode it as the following clause:

  $$(\beta_v^\tau)$$

  For example, if τ is τ_1, the top type in T, the encoding is a single clause:

  $$GETSATID(n, 4, \tau_1)$$

  For other cases, v cannot be strict subtype of τ. So for each τ's strict subtype σ, we generate a single clause

  $$(\neg\beta_v^\sigma)$$

  to express the subtype constraint.

  For example, if τ is τ_2 in type T, the encoding would be:

  $$\neg GETSATID(n, 4, \tau_4)$$

- Case three ($v_1 <: v_2$):

  If a variable v_1 is subtype of another variable v_2, then we need to go through all the type qualifiers in the underlying type system, enumerate all the possibilities, and generate the clauses for each of them.

  For each type qualifier τ_i, if it's the type qualifier for v_1, then v_2 cannot be annotated by any τ_i's strict subtype τ_j. So for each type qualifier τ_i, and each τ_i's strict subtype τ_j, we generate following clauses for them:

  $$(\neg\beta_{v_1}^{\tau_i} \vee \neg\beta_{v_2}^{\tau_j}) \wedge (\beta_{v_1}^{\tau_i} \vee \beta_{v_2}^{\tau_j})$$

  For example, in type system T, for subtype constraint $n_1 <: n_2$ since n_1 is subtype of n_2, if n_1 is τ_1, n_2 cannot be τ_2, τ_3, or τ_4; if n_1 is τ_2, n_2 cannot be τ_3 or τ_4; if n_1 is τ_3, n_2 cannot be τ_2 or τ_4; if n_1 is τ_4, n_2 might be any qualifiers in T. So the encoding for subtype constraint $n_1 <: n_2$ would be:

  $$\begin{aligned}
  &\big(\neg GETSATID(n_1,4,\tau_1) \vee \neg GETSATID(n_2,4,\tau_2)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_2) \vee GETSATID(n_1,4,\tau_1)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_1) \vee \neg GETSATID(n_2,4,\tau_3)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_3) \vee GETSATID(n_1,4,\tau_1)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_1) \vee \neg GETSATID(n_2,4,\tau_4)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_4) \vee \big(GETSATID(n_1,4,\tau_1)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_2) \vee \neg GETSATID(n_2,4,\tau_3)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_3) \vee GETSATID(n_1,4,\tau_2)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_2) \vee \neg GETSATID(n_2,4,\tau_4)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_4) \vee GETSATID(n_1,4,\tau_2)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_3)\big) \vee \big(\neg GETSATID(n_2,4,\tau_2)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_3) \vee GETSATID(n_1,4,\tau_2)\big)\\
  &\wedge\big(\neg GETSATID(n_1,4,\tau_3) \vee \neg GETSATID(n_2,4,\tau_4)\big)\\
  &\wedge\big(GETSATID(n_2,4,\tau_4) \vee GETSATID(n_1,4,\tau_3)\big)
  \end{aligned}$$

- *Encoding for Preference Constraint*

  Preference constraint is always between a variable and a constant, so we simply treat it as an equality constraint, and add the weight when we put the clause to SAT solver. The preference constraint is a soft constraint, so the SAT solver will try to satisfy the clauses for preference constraint, if there is no satisfiable solution for all of the clauses, clauses for preference constraint is not required to be satisfied, and all other clauses have to be satisfied.

- *One-hot Encoding*

  For each slot, we have to make sure that exactly one of the type qualifiers can be annotated to the slot. The serializer also generates one-hot encodings to ensure the condition.

  For example, for slot Id n and type system T, Max-SAT solver generates following one-hot encodings.

$$\begin{aligned}
&\big(GETSATID(n,4,\tau_1) \vee GETSATID(n,4,\tau_2) \\
&\vee GETSATID(n,4,\tau_3) \vee GETSATID(n,4,\tau_4)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_1) \vee GETSATID(n,4,\tau_2)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_1) \vee GETSATID(n,4,\tau_3)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_1) \vee GETSATID(n,4,\tau_4)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_2) \vee GETSATID(n,4,\tau_3)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_2) \vee GETSATID(n,4,\tau_4)\big) \\
&\wedge\neg\big(GETSATID(n,4,\tau_3) \vee GETSATID(n,4,\tau_4)\big)
\end{aligned}$$

In Max-SAT serializer, all of the constraints are serialized to CNF formulas according to above rules.

Once all of the clauses are generated, they will be put into a SAT solver, and the SAT solver returns a truth assignment for each Boolean variable in the clauses.

3.2.2 Max-SAT Solver

Max-SAT Solver is the solver that can solve *Max-SAT* problem. The input for *SAT Solver* is a Boolean formula, and the output is a truth assignment that can make the maximum number of clauses be true. Once the formula is generated, we can use existing solvers to solve it. The solvers *sat4j*[15] and *lingeling* [5] are usually used in the Max-SAT back end.

Normally, the integer value "0" in SAT solver means the end of clause, and we cannot use "0" as the input value. So before putting the clauses into SAT solver, all of the numbers in clauses are incremented by 1. And when we decode the result from SAT solver, we decrement all the numbers by 1.

sat4j is chosen because it's written in Java, such that we can use its API in the code of back end directly, and it's very convenient to run multiple solvers in different threads at same time. As we mentioned in Section 3.1.1 and Section 3.1.2, sometimes we separate the constraints in to different groups, and it's very important to performance if we can solve those groups of constraint in parallel.

lingeling is written in C, so if we need to run multiple lingeling back ends in parallel, we will need to create a process for each lingeling instance.

For performance, for relatively large input, *lingeling* is faster than *sat4j*, however for smaller target program, we prefer sat4j, since it can run in the same JVM as the whole system.

The output of SAT solver is a list of signed integer. The positive and negative integers indicate the value should be assigned to true and false respectively. Then the result is sent to the decoder, and the decoder will figure out what does these signed integer represent.

3.2.3 Max-SAT Decoding

Decoder looks through all of signed integers from SAT solver. For the negative numbers, we just skip them because negative integer indicates the corresponding slot id cannot be annotated by some qualifiers, in which we are not interested, and for the positive numbers, following steps are performed:

First, as we mentioned in Section 3.2.2, all of the numbers in CNF formula are incremented by 1, so the first step is decrementing the numbers by 1.

Second step is dividing the positive number by the number of type qualifiers in type system. For type system T in Figure 3.8, the divisor is 4. The result of division are a quotient q and a remainder r. $q + 1$ is the corresponding slot id n, and r corresponds to one type qualifier τ, which can be found in the mapping table (Figure 3.9). Then we know that the slot n should be annotated by the qualifier τ.

After iterating all of the signed integers, we can map every slot id to a type qualifier, which should be the final inference result and returned to front end.

3.3 LogiQL Back End

This section explains LogiQL back end in detail. In Section 3.1.1 and 3.1.4 we mentioned that the infrastructure of Type Constraint Solver has extendability that can be extended by other back ends. LogiQL back end described in this section will show how we implement another back end in the Type Constraint Solver. LogiQL encoding is an attempt to encode type constraint in LogiQL language. The LogiQL encoding is inspired by Max-SAT encoding. For each constraint we enumerate all the possible situations for type variables in underlying type system, and let LogicBlox calculate the corresponding type qualifier.

Section 3.3.1 shows how to encode all kinds of constraint as statements of LogiQL language. Section 3.3.2 discusses how the back end interacts with LogicBlox database. Section 3.3.3 describes how to transform the output from LogicBlox to the type qualifier for each slot.

3.3.1 LogiQL Encoding

This section discusses how to encode constraints as LogiQL language. This transformation is done by LogiQL serializer.

LogiQL is a declarative logic programming language developed by LogicBlox, Inc. to harness the power of first-order logic to support access to databases. It has been influenced by two other logic languages, Prolog and Datalog[23].

We encode constraints as LogiQL for two reasons. First, the concept of slots can be easily represented by the *entity type* in LogiQL. In LogiQL, an entity is a concrete object or abstract concept, which can be described with one or more values[23]. In *Checker Framework Inference* system, every slot corresponds to a unique integer ID, which could be used as the key for the slot entity. The declaration of slot entity in LogiQL would be:

```
Variable(v), hasVariableName(v:i) -> int(i).
```

Then we can apply *delta modifier* ”+” to insert variables in the entity:

```
+Variable(v), +hasVariableName[v] = 20.
```

Note that we use the name ”Variable” here as the entity's name because this entity only stores *variable slot*, and we use another entity ”Constant” to store *constant slot*:

```
Constant(c), hasConstantName(c:s) -> string(s).
```

The usage of these two entities will be explained later.

Second, the constraint system, which represents the relationship between slots can be expressed by *functional predicates* and *declaring derivation rules* in LogiQL. In LogiQL, functional predicates can express the association between different entities. For example:

```
equalityConstraint(v1, v2) -> Variable(v1), Variable(v2).
```

The above functional predicates means there is an equality constraint between variable slot v_1 and v_2. Similar to the variable entity, we can also use *delta modifier* to insert a new equality constraint in the predicate:

```
+equalityConstraint(v1, v2),
    +Variable(v1), +hasVariableName[v1] = 15,
    +Variable(v2), +hasVariableName[v2] = 20.
```

In LogiQL, some facts are simply asserted to be true, while other facts are computed by applying a declaring derivation rule to facts are already known. And in our LogiQL encoding, we use different declaring derivation rules to compute the type modifier for each slot. For example, if we still consider about the type hierarchy of type T in Figure 3.8, we will have one following declaring derivation rule:

```
isT1[v2] = true <- equalityConstraint(v1, v2), isT1[v1] = true.
```

which means if there is a an equality constraint between v_1 and v_2, and the type qualifier of v_1 is τ_1, then the type qualifier of v_2 is also τ_1. The declaration of functional predicates "isT1" is:

```
isT1[v] = i -> Variable(v), boolean(i).
```

For each type qualifier in type system, we have above predicate for it, and the boolean value indicates whether that variable slot should be annotated by the corresponding type type qualifier. All of these entities, predicates and derivation rules will be explained later in detail.

The logic of current logiQL encoding is very similar to the logic of Max-SAT encoding, and Boolean clauses are expressed by LogiQL predicates and derivation rules. Current

LogiQL encoding uses LogicBlox in a low level, and doesn't harness the full power of LogicBlox. In Chapter 5 we will show the performance of LogiQL back end. Compare to Max-SAT back end, it takes much more time than Max-SAT back end. In the future, we would like to explore more power of LogiQL language so that for very large programs, the performance of LogiQL encoding can be more competitive.

Similar to the Max-SAT encoding, the LogiQL encoding for *Subtype Constraint*, *Equality Constraint*, *Inequality Constraint*, *Comparable Constraint*, and *Preference Constraint* can be generated automatically according to the type hierarchy, but for *Adaptation Constraint*, the encoding has to be made by overriding the *serialize* method for *Adaptation Constraint* in *ConstraintSerializer* manually.

The rest of this section explains the LogiQL encoding in detail, and we will use the type hierarchy of type T in Figure 3.8 as an example.

- *Basic Encoding*

  Basic encoding contains the basic entity types, functional predicates and declaring derivation rules, which are not related to constraints. As we mentioned before, we used two entities to store all of the variables and constants:

  ```
  Variable(v), hasVariableName(v:i) -> int(i).
  Constant(c), hasConstantName(c:s) -> string(s).
  ```

  All of the constants will be added into Constant entity by delta modifier ”+” at the very beginning (”t” could represent any constant):

  ```
  +Constant(c), +hasConstantName[c] = "t".
  ```

  In type system T, there are four constants: τ_1, τ_2, τ_3, and τ_4, so we have four following delta predicate:

  ```
  +Constant(c), +hasConstantName[c] = "t1".
  +Constant(c), +hasConstantName[c] = "t2".
  +Constant(c), +hasConstantName[c] = "t3".
  +Constant(c), +hasConstantName[c] = "t4".
  ```

  Normally variables are added to the entity together with constraint.

  We will need the following functional predicate, which indicates the annotation(type modifier) for each variable, and the final calculated result will be store in this predicate.

  ```
  annotationOf[v] = s -> Variable(v), string(s).
  ```

For each type qualifier, we have one functional predicate that indicates whether a variable should be annotated by the corresponding type modifier. All the predicates are in form:

```
isT[v] = b -> Variable(v), boolean(b).
```

In type system T, there are four functional predicates for all the type qualifiers:

```
isT1[v] = b -> Variable(v), boolean(b).
isT2[v] = b -> Variable(v), boolean(b).
isT3[v] = b -> Variable(v), boolean(b).
isT4[v] = b -> Variable(v), boolean(b).
```

For the boolean value for a variable in "isT[v]" predicate, if the value is true, then the annotation for that variable should be "T". We use following declaring derivation rule to ensure the above condition:

```
annotationOf[v] = "t" <- isT[v] = true.
```

For type system T, the rules are written below.

```
annotationOf[v] = "t1" <- isT1[v] = true.
annotationOf[v] = "t2" <- isT2[v] = true.
annotationOf[v] = "t3" <- isT3[v] = true.
annotationOf[v] = "t4" <- isT4[v] = true.
```

The first rule can be read as "in isT1 predicate, if v corresponds to a boolean value true, then in annotationOf predicate, v will correspond to string "t1"", and same for other predicates. The annotationOf predicate is designed for making the decoding process easier. Instead of going through all the truth values in predicates "isT[v]", the "annotationOf" predicate collects all the result from those "isT[v]" predicates.

Besides above predicates, we need to make sure that only one of type qualifiers can be annotated to each slot. In logiQL encoding, we use following declaring derivation rules to ensure that condition:

```
isT2[v] = false <- isT1[v] = true.
isT3[v] = false <- isT1[v] = true.
isT4[v] = false <- isT1[v] = true.
isT1[v] = false <- isT2[v] = true.
isT3[v] = false <- isT2[v] = true.
isT4[v] = false <- isT2[v] = true.
isT1[v] = false <- isT3[v] = true.
isT2[v] = false <- isT3[v] = true.
isT4[v] = false <- isT3[v] = true.
isT1[v] = false <- isT4[v] = true.
```

```
        isT2[v] = false <- isT4[v] = true.
        isT3[v] = false <- isT4[v] = true.
```

That's all for basic encoding, these predicates and entities provide basic knowledge of constraints and type modifiers, and will be used in the encoding kinds of.

- *Encoding for Equality Constraint*

  The two basic predicates for equality constraint are:

  ```
  equalityConstraint(v1,v2) ->
      Variable(v1), Variable(v2).
  equalityConstraintContainsConstant(c,v) ->
      Constant(c), Variable(v).
  ```

  The first predicate expresses that there is an equality constraint between variable v_1 and variable v_2. The second predicate express the equality constraint is between a constant and a variable.

  For each type qualifier in the type system, we use two rules for equality constraint related calculation:

  ```
  isT[v2] = true <-
      equalityConstraint(v1, v2), isT[v1] = true.
  isT[v1] = true <-
      equalityConstraint(v1, v2), isT[v2] = true.
  isT[v2] = false <-
      equalityConstraint(v1, v2), isT[v1] = false.
  isT[v1] = false <-
      equalityConstraint(v1, v2), isT[v2] = false.
  isT[v] = true <-
      equalityConstraintContainsConstant(c, v),
      hasConstantName(c:"t").
  ```

  The first rule can be read as "if there is an equality constraint between v_1 and v_2, and the type qualifier for v_1 is τ, then the type qualifier for v_2 is τ". The second rule can be read as "if there is an equality constraint between constant τ and variable v, then the type qualifier for v is τ". So for equality constraint, the number of rules is twice as much as the number of type qualifiers.

  For type system T, we generate following rules for equality constraint:

  ```
  isT1[v2] = true <-
      equalityConstraint(v1, v2), isT1[v1] = true.
  isT1[v1] = true <-
      equalityConstraint(v1, v2), isT1[v2] = true.
  ```

```
isT1[v2] = false <-
    equalityConstraint(v1, v2), isT1[v1] = false.
isT1[v1] = false <-
    equalityConstraint(v1, v2), isT1[v2] = false.
isT1[v] = true <-
    equalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t1").
isT2[v2] = true <-
    equalityConstraint(v1, v2), isT2[v1] = true.
isT2[v1] = true <-
    equalityConstraint(v1, v2), isT2[v2] = true.
isT2[v2] = false <-
    equalityConstraint(v1, v2), isT2[v1] = false.
isT2[v1] = false <-
    equalityConstraint(v1, v2), isT2[v2] = false.
isT2[v] = true <-
    equalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t2").
isT3[v2] = true <-
    equalityConstraint(v1, v2), isT3[v1] = true.
isT3[v1] = true <-
    equalityConstraint(v1, v2), isT3[v2] = true.
isT3[v2] = false <-
    equalityConstraint(v1, v2), isT3[v1] = false.
isT3[v1] = false <-
    equalityConstraint(v1, v2), isT3[v2] = false.
isT3[v] = true <-
    equalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t3").
isT4[v2] = true <-
    equalityConstraint(v1, v2), isT4[v1] = true.
isT4[v1] = true <-
    equalityConstraint(v1, v2), isT4[v2] = true.
isT4[v2] = false <-
    equalityConstraint(v1, v2), isT4[v1] = false.
isT4[v1] = false <-
    equalityConstraint(v1, v2), isT4[v2] = false.
isT4[v] = true <-
    equalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t4").
```

The meaning of rules are same as the first two rules except for applying on different constant. These rules and predicates interpret equality constraint in a very straightforward way. For example, if we have two following equality constraints (The numbers in parenthesis are slot ids):

$$v_1(1) = v_2(2)$$
$$v_2(2) = \tau_1$$

Then we will add following facts to the entities:

```
+equalityConstraint(v1, v2),
    +Variable(v1), +hasVariableName[v1] = 1,
    +Variable(v2), +hasVariableName[v2] = 2.
+equalityConstraintContainsConstant(c, v),
    +Constant(c), +hasConstantName[c] = "t1",
    +Variable(v), +hasVariableName[v] = 2.
```

The calculation starts from the fact:

```
+equalityConstraintContainsConstant(c, v),
    +Constant(c), +hasConstantName[c] = "t1",
    +Variable(v), +hasVariableName[v] = 2.
```

Then according to the rule:

```
isT1[v] = true <-
    equalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t1").
```

we can get:

```
isT1[1] = true
```

The annotation for v_2 will be calculated in similar way, and after the calculation, we will get the expected result:

```
isT1[1] = true
isT1[2] = true
```

- *Encoding for Inequality Constraint*

  Similar to the encoding for equality constraint, we have following two basic predicates for inequality constraint:

```
inequalityConstraint(v1,v2) ->
    Variable(v1), Variable(v2).
inequalityConstraintContainsConstant(c,v) ->
    Constant(c), variable(v).
```

They express the inequality constraint between variable and variable, and constant and variable respectively.

In general, we have two following rules for each type qualifier:

```
isT[v2] = false <-
    inequalityConstraint(v1, v2), isT[v1] = true.
isT[v1] = false <-
    inequalityConstraint(v1, v2), isT[v2] = true.
isT[v] = false <-
    inequalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t").
```

Still similar to equality constraint, the first rule can be read as "if there is an inequality constraint between v_1 and v_2, and the type qualifier for v_1 is τ, then the type qualifier for v_2 is not τ". The second rule can be read as "if there is an inequality constraint between constant τ and variable v, then the type qualifier for v is not τ". The number of rules is also twice as much as the number of type qualifier.

We have following declaring derivation rules for type system T:

```
isT1[v2] = false <-
    inequalityConstraint(v1, v2), isT1[v1] = true.
isT1[v1] = false <-
    inequalityConstraint(v1, v2), isT1[v2] = true.
isT1[v] = false <-
    inequalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t1").
isT2[v2] = false <-
    inequalityConstraint(v1, v2), isT2[v1] = true.
isT2[v1] = false <-
    inequalityConstraint(v1, v2), isT2[v2] = true.
isT2[v] = false <-
    inequalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t2").
isT3[v2] = false <-
    inequalityConstraint(v1, v2), isT3[v1] = true.
isT3[v1] = false <-
    inequalityConstraint(v1, v2), isT3[v2] = true.
isT3[v] = false <-
    inequalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t3").
isT4[v2] = false <-
    inequalityConstraint(v1, v2), isT4[v1] = true.
isT4[v1] = false <-
    inequalityConstraint(v1, v2), isT4[v2] = true.
```

```
isT4[v] = false <-
    inequalityConstraintContainsConstant(c, v),
    hasConstantName(c:"t4").
```

So for inequality constraints

$$v_1(1) \neq \tau_1$$

following facts will be added to entities:

```
+inequalityConstraintContainsConstant(c, v),
    +Constant(c), +hasConstantName[c] = "t1",
    +Variable(v), +hasVariableName[v] = 1.
```

After the calculation, we can get the result:

```
isT1[1] = false
```

- *Encoding for Comparable Constraint*

  For comparable constraint, we have these two entities as well:

```
comparableConstraint(v1, v2) ->
    Variable(v1), Variable(v2).
comparableConstraintContainsConstant(c, v) ->
    Constant(c), Variable(v).
```

  The declaring derivation rules for comparable constraint depends on the type system, so for the type system that every type qualifier is comparable to each other, for example, remove one of τ_2 and τ_3 in T, then every comparable constraint should be satisfied, and we don't need to generate any declaring derivation rules for that case. However if there are incomparable types in the type system like T, then we will have following rules (τ_x and τ_y are incomparable):

```
isTX[v2] = false <-
    comparableConstraint(v1, v2), isTY[v1] = true.
isTX[v1] = false <-
    comparableConstraint(v1, v2), isTY[v2] = true.
isTX[v] = false <-
    comparableConstraintContainsConstant(c, v),
    hasConstantName(c:"ty").
```

  The meaning of these rules are straightforward: if there is a comparable constraint between two variables or a variable and a constant, then τ_x and τ_y cannot be both annotated. For a type system, if there are n pairs of incomparable type qualifiers, then the number of rules is twice of that number.

  For type system T the following rules are generated:

```
isT3[v2] = false <-
    comparableConstraint(v1, v2), isT2[v1] = true.
isT3[v1] = false <-
    comparableConstraint(v1, v2), isT2[v2] = true.
isT3[v] = false <-
    comparableConstraintContainsConstant(c, v),
    hasConstantName(c:"t2").
isT2[v2] = false <-
    comparableConstraint(v1, v2), isT3[v1] = true.
isT2[v1] = false <-
    comparableConstraint(v1, v2), isT3[v2] = true.
isT2[v] = false <-
    comparableConstraintContainsConstant(c, v),
    hasConstantName(c:"t3").
```

For comparable constraints

$$v_1(1) <:> \tau_2$$

we add following facts to entities:

```
+comparableConstraintContainsConstant(c, v),
    +Constant(c), +hasConstantName[c] = "t2",
    +Variable(v), +hasVariableName[v] = 1.
```

And according to the facts and rules, we can get the fact

```
isT3[1] = false
```

- *Encoding for Subtype Constraint*

  The encoding for subtype constraint is more complicated than others because the positions of two slots in subtype constraint are unchangeable. For example $v_1 <: \tau_1$ is not equivalent to $\tau_1 <: v_1$, so we need one more entity to store the subtype constraint between variable and constant. The entities for subtype constraint are:

  ```
  subtypeConstraint(v1, v2) ->
      Variable(v1), Variable(v2).
  subtypeConstraintLeftConstant(c, v) ->
      Constant(c), Variable(v).
  subtypeConstraintRightConstant(v, c) ->
      Variable(v), Constant(c).
  ```

  We will use subtypeConstraintLeftConstant entity to store the subtype constraint where the subtype type is constant, and use subtypeConstraintRightConstant if the constant is supertype.

In subtype constraint, there are two special cases. One is the supertype is the bottom type in type hierarchy, and another is the subtype is the top type in type hierarchy. For the former case, the subtype has to be the bottom type, and for the latter case, the super type has to be the top type. In logiQL encoding, we use following rules to handle the two cases (τ_{top} and τ_{bottom} are top and bottom qualifier separately).

```
isTTOP[v2] = true <-
    subtypeConstraint(v1, v2), isTTop[v1] = true.
isTTOP[v] = true <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"ttop").
isTBOTTOM[v1] = true <-
    subtypeConstraint(v1, v2), isTBOTTOM[v2] = true.
isTBOTTOM[v] = true <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"tbottom").
```

For type system T, τ_1 and τ_4 are top and bottom qualifier:

```
isT1[v2] = true <-
    subtypeConstraint(v1, v2), isT1[v1] = true.
isT1[v] = true <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t1").
isT4[v1] = true <-
    subtypeConstraint(v1, v2), isT4[v2] = true.
isT4[v] = true <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"t4").
```

For example, for the subtype constraint

$$\tau_1 <: v_1(1)$$

We will add fact

```
+subtypeConstraintLeftConstant(c, v),
    +Constant(c), +hasConstantName[c] = "t1",
    +Variable(v), +hasVariableName[v] = 1.
```

to entities. And according to rule

```
isT1[v] = true <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t1").
```

we can get

```
isT1[1] = true
```

For other cases, we use the similar logic in Max-SAT encoding to convert subtype constraint to LogiQL. For a subtype constraint (n_1 and n_2 could be either constant or variable)

$$n_1(1) <: n_2(2)$$

we go through all the type qualifiers in type system and generate the rules for each of them. Since n_1 is the subtype of n_2, which means they are comparable, so if τ_x and τ_y are incomparable, we will have:

```
isTX[v2] = false <-
    subtypeConstraint(v1, v2), isTY[v1] = true.
isTX[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasConstantName(c:"ty").
isTX[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasConstantName(c:"ty").
```

And we have to make sure n_2 cannot be the subtype of n_1(τ_p represents any qualifier and τ_q represents any subtype of each τ_p):

```
isTQ[v2] = false <-
    subtypeConstraint(v1, v2), isTP[v1] = true.
isTQ[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"tp").
isTP[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"tq").
```

We use following rules to handle all of these situations for type system T.

```
isT4[v2] = false <-
    subtypeConstraint(v1, v2), isT2[v1] = true.
isT4[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t2").
isT4[v2] = false <-
    subtypeConstraint(v1, v2), isT3[v1] = true.
isT4[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t3").
```

```
isT3[v2] = false <-
    subtypeConstraint(v1, v2), isT2[v1] = true.
isT3[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t2").
isT2[v2] = false <-
    subtypeConstraint(v1, v2), isT3[v1] = true.
isT2[v] = false <-
    subtypeConstraintLeftConstant(c, v),
    hasconstantName(c:"t3").
isT3[v1] = false <-
    subtypeConstraint(v1, v2), isT2[v2] = true.
isT3[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"t2").
isT1[v1] = false <-
    subtypeConstraint(v1, v2), isT2[v2] = true.
isT1[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"t2").
isT2[v1] = false <-
    subtypeConstraint(v1, v2), isT3[v2] = true.
isT2[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"t3").
isT1[v1] = false <-
    subtypeConstraint(v1, v2), isT3[v2] = true.
isT1[v] = false <-
    subtypeConstraintRightConstant(v, c),
    hasconstantName(c:"t3").
```

- *Encoding for Preference Constraint*

  At the moment, we don't have an effective way to encode preference constraint, but we have an option to treat preference as equality constraint. In that case, for preference constraint

  $$v_1 \sim= c_1$$

  c_1 will be the type qualifier for v_1. Finding a effective LogiQL encoding for preference constraint is a future work to enhance the functionality of LogiQL back end.

All of entities, functional predicates, and declaring derivation rules are generated according to the type hierarchy of the type system. And all of the facts that applying "+"

delta modifier are generated according to constraints. Once entities, predicates, rules, and facts are generated, logiQL back end puts them into LogicBlox in a separate process.

3.3.2 LogiQL Solver

In LogiQL back end, the solver is LogicBlox, which calculates the results for different predicates according to facts in entities and rules. The back end calls LogicBlox by creating a new process. LogicBlox is invoked by command ”lb”, and normally following commands are executed:

```
lb create myworkspace
lb addblock myworkspace -f tables.logic
lb exec myworkspace -f facts.logic
lb print myworkspace annotationOf
lb delete myworkspace
```

First and fifth commands creates and deletes the workspace respectively. Second command adds all necessary entities, predicates, rules in LogicBlox, and third one put all facts in logicBlox. Forth command prints the results in *annotationOf*. Then the back end reads the output, and send it to the decoder. File ”tables.logic” and ”facts.logic” are the logiQL encoding generated from logiQL serializer.

3.3.3 LogiQL Decoding

As we mentioned in Section 3.3.2, we use the data in predicates *annotationOf* as result.

The output from ”print” command of LogicBlox is in form:

```
3 ”t1”
5 ”t3”
```

which means the calculated type qualifier for slot 3 is τ_1, and the qualifier for slot 5 is τ_3. The decoder reads this output line by line, the number before the middle space is slot id, and the string after the space is the name of type qualifier.

After reading all of the output, we can construct a map between each slot id and a type qualifier, then send the map back to front end.

Chapter 4

Dataflow Type System

This chapter discusses an inferable type system called Dataflow. Dataflow type system and its inference is used for data-flow analysis. We use Dataflow as an example type system to validate the correctnesses of our type inference tool. Note that the data-flow analysis that the type system performs only focuses on the type of each allocation rather than the data values or the sites of allocations. Section 4.1 provides a basic overview of Dataflow type system. Section 4.2 discusses the simplification process for Dataflow type qualifier, which is a key process for type inference. Section 4.3 explains the type hierarchy of Dataflow type system. Section 4.4 explains the type inference approach for Dataflow, and how to integrate the approach into the back end structure.

4.1 Introduction on Dataflow

In Java, the type of fields, methods, parameters, and local variables in run time sometimes is not exact same as the declared type because of the polymorphism concept. For example:

```
Object foo(boolean bool) {
  if(bool) {
    return "str";
  }
  return new Date();
}
```

The declared return type of method foo is ”Object”, but at run time, the real return type would be either ”String” or ”Date”. In static analysis, sometimes we are interested in comparing the similarity between two pieces of code. One of similarity aspects is the type of input and output, and for methods, they are the type of parameters and return type respectively. However, the declared return type in method head, like ”Object” in foo, cannot provide enough information to us. Knowing more details of those types would be helpful for static analysis. We introduce Dataflow type system that can help us to know run-time types of fields, methods, parameters, and local variables in detail.

We use a single Java annotation *@Dataflow* with two arguments *typeNames* and *typeNameRoots* to represent all of the Dataflow type qualifiers. Value in argument *typeNames* and *typeNameRoots* is a sequence of fully qualified Java type names. For example, ”java.lang.String, java.lang.Integer”.

- *typeNames*

  Values in *typeNames* indicate that all possible Java run-time types that may be returned by the annotated method, or be assigned to the annotated variable, parameters, and fields.

- *typeNameRoots*

  Values in *typeNameRoots* indicate all possible upper bounds of run-time types of method return types rather than exact Java types. The value of *typeNameRoots* will be set if a method from byte code is called.

Both arguments *typeNames* or *typeNameRoots* could be empty, in that case, the Dataflow qualifier indicates that we don't have any knowledge about the type. A Dataflow type will be generated with typeNames value for the expressions that create new types, like *Class Instance Creation Expression* , *Array Creation Expression* and *Literals.* For example:

```
int x = 0;
new Object();
new int[1];
```

The qualifier for right hand side of first statement would be *@DataFlow(typeNames=”int”)*, and the qualifier for the second and third statement would be *@DataFlow(typeNames=”java.lang.Object”)* and *@DataFlow(typeNames=”int[]”).*

If a method from byte code is invoked, then the Dataflow type with typeNameRoots value will be generated for that *MethodInvocation* expression. For example:

```
(new StringBuilder("a")).toString();
```

For the above statement, the Dataflow type for it would be *@DataFlow(typeNameRoots=”java.lang.String”)*. We need *typeNameRoots* values because we cannot see the method body if the method is pre-compiled, and the declared type in method head is the only knowledge about the method's return type. However, in run time there still may be multiple possibilities for the return type. So the word ”root” indicates that the type is the upper bound of real return types.

For now, the only use of TypeNameRoots is in method invocation from Java byte code, however, this arguments can also be used when we read a field from Java bytecode. But in current implementation, the system won't annotate the case that accesses to fields declared in bytecode.

A Dataflow type qualifier is supposed to contain all possible run-time types for the annotated location, however, the current inference approach for Dataflow type system is limited for some situations, for example, if a method from bytecode accesses a field, the inferred result would be very conservative. This will be explained in Section 4.4 in detail.

4.2 Type Simplification

In *@Dataflow* qualifier, values in *typeNames* represents a precise Java type, and the *typeNameRoots* represents the upper bound of Java types. Therefore, some dataflow types can be simplified according to the relationship among the Java types included in *typeNames* and *typeNameRoots*.

The basic idea of simplification is we don't want any overlaps among the values in typeNameRoots and typeNames. If there are some overlaps, the most general one will be kept.

For example, for a Dataflow type, if the values of *typeNameRoots* are ”java.lang.String”, ”java.lang.Number”, and ”java.lang.Short” the values of *typeNames* are ”java.lang.Byte” and ”java.lang.Object”, then since Byte and Short are both subtype of Number, and they can be upper bounded by Number, so after the simplification, the *typeNameRoots* result would be ”java.lang.String” and ”java.lang.Number”, *typeNames* result would be ”java.lang.Object”.

The algorithm for simplification dataflow type is listed in Algorithm 5. The ”comparable” in Algorithm 5 means for the two compared types, one could be a subtype of the other. After simplification, the new qualifier is equivalent to the original one. All values in *typeNameRoots* and *typeNames* are alphabetically sorted.

Algorithm 5 Algorithm for type simplification

```
let typeNamesList contains all values in typeNames
let typeNameRootsList constains all values in typeNameRoots
let typeNamesResult and typeNameRootsResult be empty sets
while typeNameRootsList is not empty do
    compare the first element ele0 with all following elements eles
    if ele0 is comparable with one of eles then
        remove the subtype from typeNameRootsList
    else
        remove ele0 from typeNameRootsList
        add ele0 to typeNameRootsResult
    end if
end while
for every element ele in typeNamesList do
    if ele is subtype of one of element in typeNameRootsResult then
        continue
    else
        add ele to typeNamesResult
    end if
end for
return typeNamesResult, typeNameRootsResult
```

4.3 Dataflow Type Hierarchy

Normally, one Java annotation represents only one type qualifier. But *@Dataflow* can take arguments, and each *@Dataflow* with a specific group of arguments is one type qualifier, such that all qualifiers are expressed by a single Java annotation *@Dataflow*.

The basic idea of subtype relationship in Dataflow type system is: Dataflow type τ_1 is subtype of Dataflow type τ_2, if values in typeNames and typeNameRoots of τ_1 can be bounded by the values in τ_2. In Java, *Object* is the super type of all other reference type, so *@DataFlow(typeNameRoots=”Object”)* is the top type of Dataflow type system, and the bottom type is the qualifier with empty value meaning we don’t have any knowledge about typeNames and typeNameRoots.

For example, *@DataFlow(typeNameRoots="String", "Number", typeNames="Object")* is the super type of *@DataFlow(typeNameRoots="String", typeNames="Byte")*. Figure 4.1 shows the relationship among different types. Note that the dashed arrow is used in Figure 4.1 because the relationship between each pair of qualifier is not strict subtype.

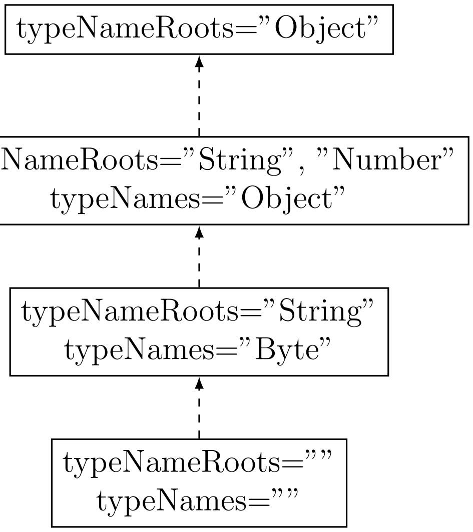

Figure 4.1: Type Hierarchy of Dataflow Type System

The calculation of subtype relationship in Dataflow type system is based on the type simplification. The approach for deciding whether τ_1 is subtype of τ_2 is: we concatenate the values in typeNames of two qualifiers together, and do same thing for typeNameRoots, then call the simplification algorithm for the two concatenated lists. If the simplified result is identical with the values in τ_2, then τ_1 is subtype of τ_2.

For the types that have both primitive and reference type representation, like *int* and *java.lang.Integer*, the name we are going to choose depends on whether the type is autoboxed or unboxed. If autoboxing or unboxing process is called, we will choose the name after autoboxing and unboxing.

Since *@DataFlow(typeNameRoots="Object")* is the top type of *Dataflow Type System*, according to the type hierarchy, every location can be annotated by *@DataFlow(typeNameRoots="Object")* without breaking type rules, however, we would like to know all concrete possible run-time types for a given location. Since the possible run-time types of each location are set once the program had been written, there should always be a unique best Dataflow type qualifier for the location. The details of inferring the best solution will be discussed in Section 4.4.

4.4 Type Inference for Dataflow

This section describes the type inference approach for Dataflow type system. First we introduce the basic idea of inference approach, then we describe how to integrate the approach with the *Type Constraint Solver* structure.

The ultimate goal for Dataflow type system is inferring all possible run-time types for return types, parameters, fields, and variables at compile time, however the current inference approach is too conservative for some cases. For example, if a method from Java type code writes values to a field in source code, then current implementation will infer a TypeNameRoot for it. In this case, we may lose lot of information about the field. So the limitation of current approach is we can only infer the run-time types we can observe.

4.4.1 Annotate Base Cases

The Dataflow annotation for variables and methods represents all the possible Java types for them. In order to determine the Dataflow annotations, we first look at all the places that the ”base case” is created. Base case are the Java expressions that we can know the Java types of them directly. As we mentioned in the beginning of this chapter, base cases are: *Literals*, *Class Instance Creation Expression*, *Array Creation Expression*, and *Method Invocation Expression* for the method from byte code. We look through all these base cases, and create Dataflow annotations for them as start points of inference process. In Checker Framework Inference system, each tree node corresponds to a slot, and since the annotations for these base cases are fixed, we create constant slots with the Dataflow annotations for these cases.

4.4.2 Constraint Generation

This step is described in [18] in detail. For given program, Checker Framework Inference system creates a set of constraints, and the slots for base cases are annotated by Dataflow annotations.

Besides annotating base cases, another dataflow specific implementation in this step is that we handled autoboxing and unboxing for primitive types. For example, if we have following variable declaration:

```
1 Integer int1 = 1;
2 int int2 = new Integer(1);
```

We want the integer literal 1 in line 1 to be annotated by *@DataFlow(typeNames=”java.lang.Integer”)*, because the Java compiler runs autoboxing mechanism on it, so that it should be used as a reference type Integer. The annotation for new Integer(1) in line 2 is *@DataFlow(typeNames=”int”)* for same reason. The system checks all the places that autoboxing and unboxing happens, and creates corresponding Dataflow annotations for those places.

4.4.3 Constraint Separation

Once the constraint set is generated, we need to solve them and then get the concrete annotations for all slots. There is a significant property in Dataflow type system: the presence of one Java type in Dataflow annotation is independent with other types. For example, an object is assigned with a new String doesn't affect we assign a new HashMap to it later. This property allows us to group the constraints by different types, solve them independently, and merge them in the end. Note that grouping the constraints by different types before solving them is a necessary step, otherwise, the solver may provide some unexpected results. The reason of why constraint separation is a necessary step will be explained in Section 4.4.4.

A correct separation is important, for each base case, we want to see the case could have an influence on which slots, so we take advantage of *Constraint Graph* at this step. In Constraint Graph, we have an option to get the different sets of constraints grouped by constant slot. For example, if we have following code:

```
Integer i = new Integer(1);
String str = "s";
Object obj1 = i;
obj1 = s;
Object obj2 = obj1;
```

Then the string literal *”s”* and the *new Integer(1);* are bases cases, we create two constant slots with the value *@DataFlow(typeNames=”java.lang.Integer”)* and *@DataFlow(typeNames=”java.lang.String”)* respectively. And we would get at least five subtype constraints for the five statements:

```
@DataFlow(typeNames={"java.lang.Integer"}) <: i
@DataFlow(typeNames={"java.lang.String"}) <: str
i <: obj1
s <: obj1
obj1 <: obj2
```

In above constraints, we use variable’s name instead of slot id for the sake of convenience. Figure 4.2 shows the constraint graph for example code. From the graph, we know that base case *@DataFlow(typeNames=”java.lang.Integer”)* has influence on variable i, obj1 and obj2. *@DataFlow(typeNames=”java.lang.String”)* has influence on variable str, obj1 and obj2. So we group the constraints into two sets by the two Java type:

```
java.lang.Integer:
```

```
@DataFlow(typeNames={"java.lang.Integer"}) <: i
i <: obj1
obj1 <: obj2
```

```
java.lang.String:
```

```
@DataFlow(typeNames={"java.lang.String"}) <: str
s <: obj1
obj1 <: obj2
```

In next step, we will solve them independently.

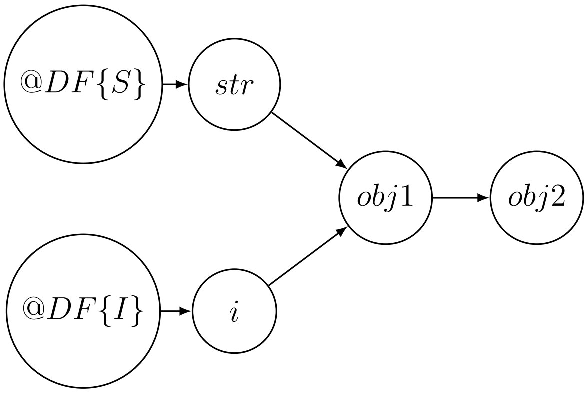

Figure 4.2: Constraint Graph for Example Code

4.4.4 Constraint Solving

After the above step, each Java type corresponds to a group of constraint, and the question is for the slots in each group of constraint, whether the corresponding Java type should be present in them or not. The solution for the problem is when we solve each group of constraints, instead of Dataflow itself, we treat the underlying type system as a two qualifiers type system that only contains top and bottom qualifiers. The top qualifier is *@DataFlow*, and the value is the corresponding Java type for current group of constraint like *@DataFlow(typeNames="int")*, and the bottom qualifier is *@DataFlow* without any values. For example, for the constraints showed in section 4.4.3, the five subtype constraints will be solved twice, once for Integer and once for String. When we solve for type Integer, we only look at constraint 1, 3 and 5, and the two qualifiers type system is used. Top qualifier is *@DataFlow(typeNames="java.lang.Integer")*, and bottom qualifier *@DataFlow()*. After the back end solves constraints and gives the solution, we only keep tracking the slots that are annotated by the top qualifier. Note that *@DataFlow(typeNames="java.lang.Integer")* and *@DataFlow(typeNameRoots="java.lang.Integer")* are two different annotations so that "java.lang.Integer" has different meanings. So if the DataFlow annotation with typeNameRoots value shows in the constraint, it will be a constant as well, and corresponds to some constraints.

The reason why this approach works is, first, as we mentioned before, the presence of different Java types are independent with each other, so for a group of constraint, we can only care about the presence of the corresponding type. Second, Dataflow's type hierarchy defines that if a certain Java type is present in one Dataflow type qualifier, then the Java type has to be able to be bounded by it's super type. That's why we put the annotation that we care about in top: the supertype of the top qualifier has to be the top qualifier as well, which means the Java type presenting in top qualifier has to be ”present” in the supertype of top qualifier. The ”present” here has two cases: one case is the Java type is present in the super type physically, and another is it's bound by one of the *typeNameRoots* in super type. No matter which situation happens, there is no problem of doing this at this step, because we will run simplification algorithm when we merge the solution. So if second situation happens, the Java type that we put in the Dataflow qualifier at this step will be removed eventually. The merge approach will be explained in section 4.4.5 in detail.

In Section 4.4.3 we mentioned that we have to separate the constraints before solving them. Thinking about following Java code:

```
Integer i1 = new Integer(1);
String str1 = "s";
Integer i2 = i1;
String str2 = str1;
```

We would get at least four subtype constraints for the four statements:

```
@DataFlow(typeNames={"java.lang.Integer"}) <: i1
@DataFlow(typeNames={"java.lang.String"}) <: str1
i2 <: i1
str2 <: str1
```

Since there are two constant values: *@DataFlow(typeNameRoots=”java.lang.Integer”)* and *@DataFlow(typeNameRoots=”java.lang.String”)*, they will be solved twice. However, if we solve all of them at same time, when we solve for type Integer, the serializer still generate the encoding for the constraint $str2 <: str1$ as well, and any random result given by the solver will satisfy the constraint because we only focus on type Integer now. In other words, the solver may randomly make a decision about whether the Dataflow type qualifier for $str2$'s type variable should contain *typeNames=”java.lang.Integer”* or not. But in the program, $str2$ is only related to java.lang.String. So the constraint separation step can successfully make constraint $str2 <: str1$ unvisitable to serializer when it generates the encoding for type Integer.

After solving the different group of constraints, we can get a set of solution, and every solution corresponds to a particular Java type as each group of constraints does. In each solution, we know which slot should contain the corresponding Java type. Now, we are ready to merge the solutions together.

4.4.5 Solution Merge

In this step, the solutions from different group are going to be merged, and the final result is generated. The result for one slot may be partially apparent in different solution. For example, in section 4.4.3, the final Dataflow annotations for variable *obj1* and *obj2* should contain both *java.lang.String* and *java.lang.Integer*, but this knowledge is in the two solutions. The merge algorithm is straightforward: we go through every solution, put the results for each slot in different solution together, and then run the simplification algorithm on each merged annotation. The output from simplification algorithm is the final result for each slot. If the results for one slot from different solutions only contain the annotations with *typeNames* value, then the final result would be the concatenation among all *typeNames* values. However, if *typeNameRoots* is involved, then some *typeNames* or *typeNameRoots* values may be removed after simplification according to the relationship among those Java types.

Chapter 5

Implementation and Experimentation

This Chapter describes the implementation of *Type Constraint Solver* and *Dataflow Type System* (Section 5.1), and experience with them (Section 5.2).

5.1 Implementation

The *Type Constraint Solver* described in Chapter 3 and *Dataflow Type System* described in Chapter 4 are both implemented on top of *Checker Framework Inference* [1], which has been introduced in Section 2.1.

Type Constraint Solver and *Dataflow Type System* consist of about 4400 and 1220 non-comment, non-blank lines of Java code respectively. *Checker Framework Inference* provides an interface called *InferenceSolver*, it passes slots, constraints, and other useful information to *Type Constraint Solver*. *Dataflow Type System* consists of its own checker, solver, and other related components. Dataflow solver is built on the top of *Type Constraint Solver*. As we discussed in Chapter 3, the infrastructure of *Type Constraint Solver* is easy to extend in order to implement the type system specific feature.

Type Constraint Solver supports different solving options, for example, run constraint separation algorithm or not, solve the set of constraint in parallel or sequential, and the type of back end.

For *Dataflow Type System*, we integrated the inference approach into the back end structure that we introduced previously.

The most significant and necessary step of inferring Dataflow annotations is constraint separation, unlike the type systems without any arguments, all of the constraints can be solved once, we have to group the constraints by different Java type, and solve them separately. Fortunately, back end structure supports solving constraints in different groups, and we also have the option that groups the constraint by different constants in the stage of constraint graph construction. So we can create a list of *BackEnd* instances with different constraints configuration, and solve them separately.

Section 4.4.4 explains that when we solve the different groups of constraint, we treat the underlying type system as a two qualifiers type system, so the lattice configuration for different back ends are also different. We made a *TwoQualifiersLattice* class as a subclass of *Lattice* class, it takes two *AnnotationMirror*s, top and bottom, as type arguments. And the map between clause integer and *AnnotationMirror* is inherited. Before we create an instance of *BackEnd* class, we create a new instance of *TwoQualifiersLattice*, set the top annotation be the corresponding Dataflow annotation as it's explained in 4.4.4, and use it as the lattice configuration for the new back end.

For the merge step, instead of using the default merge strategy, which is simply concatenating the results from different back ends, we use the approach described in 4.4.5.

We created a new class *DataflowConstraintSolver* as the subclass of *ConstraintSolver*, and implemented the Dataflow specific behavior inside this class.

5.2 Experimentation

We applied *Type Constraint Solver* with two kinds of type systems, *OsTrusted Type System*, and *Dataflow Type System*, on six real-world open source Java projects developed by external developers. The six projects are: (1) ant-javacard, a project integrates JavaCard CAP into Ant built system, (2) jdeb, an Ant task and a Maven plugin to create Debian packages from Java builds in a truly cross platform manner, (3) JReactPhysics3D, a 3D physics engine written in Java, (4) jsoup, a Java library for working with HTML, extracting, and manipulating data, (5) dyn4j, a Java 2D collision detection and physics engine, and (6) ode4j, a Java 3D Physics Engine. These six projects are in different sizes, from around 600 lines of code to 40000 lines of code. We choose the projects in different sizes because we would like to know how the scalability as regards the performance of our tools.

Table 5.1 and Table 5.2 show the size of open source projects. Since we ran our tool with different type systems, the size of slot and constraint generated by *Checker Framework Inference* may be different. The two tables show the benchmarks' information with *Dataflow Type System* and *OsTrusted Type System* respectively. SLOC gives the

Benchmark	Project Size							
	Files	Blank	Comment	SLOC (Java)	Slot		Constraint	
					Constant Slot	Variable Slot	Subtype Constraint	Equality Constraint
ant-javacard	3	941	95	627	63	425	701	56
jdeb	73	1453	1911	5062	219	3263	5610	442
jReactPhysics3D	85	3252	5579	9466	132	6065	11386	1034
jsoup	87	3036	4518	16426	339	9856	18826	947
dyn4j	209	4164	24975	17845	286	16532	27020	2413
ode4j	234	10792	36662	39402	435	30980	54208	4409

Table 5.1: Size of Projects with Dataflow Type System Inference

number of non-black, non-comment lines as determined by the cloc tool. The constraint and slot size columns give the number of constraints variables (slot) and constraints in the program.

We ran our inference tool on those projects with different options and back ends, and measured the running time information. We executed each run three times and report the median. We did each run multiple times since we found that the variance of different executions is at 0.1 to 1 seconds level.

Table 5.3 shows the timing result of inference for Dataflow Type System by Max-SAT back end. Since the constraint separation step is mandatory for inference of Dataflow Type System, there is no timing result for solving all constraints as a whole. Table 5.4 is the timing result of inference on Dataflow Type System with LogiQL back end. For LogiQL back end, we could not run it in parallel like what we did for Max-SAT back end, since we only use one machine with LogicBlox installed. We will need to distribute different tasks to multiple machines in order to perform parallel calculation.

Table 5.5 and Table 5.6 show the timing results of inference for OsTrusted Type System by Max-SAT back end with and without constraints separation respectively. Table 5.7 is the inference result for OsTrusted Type System by LogiQL back end and with separation Algorithm 2 in Section 3.1.2. Table 5.8 is the inference result for OsTrusted Type System by LogiQL back end without any constraint separation.

We analyzed the statistics in following tables, and the explanation of the timing result is discussed below.

Benchmark	Project Size							
					Slot		Constraint	
	Files	Blank	Comment	SLOC (Java)	Constant Slot	Variable Slot	Subtype Constraint	Equality Constraint
ant-javacard	3	941	95	627	3	435	726	58
jdeb	73	1453	1911	5062	3	3547	5936	454
jReactPhysics3D	85	3252	5579	9466	3	7640	14212	1092
jsoup	87	3036	4518	16426	3	10340	19615	994
dyn4j	209	4164	24975	17845	3	16909	27805	2426
ode4j	234	10792	36662	39402	3	31632	55277	4419

Table 5.2: Size of Projects with OsTrusted Type System Inference

We will first discuses different tables respectively, and then show the comparison among tables.

- *Table 5.3: Inference for Dataflow Type System by Max-SAT back end*

  We noticed that for both Sat4j and Lingeling solvers, the overall processing time in sequential is always longer than the time in parallel. Although there is overhead in creating multiple threads/processes, for the scale of the benchmarks we experienced, the speed is faster if we process different components simultaneously.

  As we mentioned in Section 3.2.2, the Lingeling solver is written in C, and it's a fast and modern SAT solver, so no matter we solve the constraint in sequential or in parallel, lingeling solver is always faster than Sat4j solver.

  If we compare the fourth row and fifth row, we can find that for Sat4j, the solving time not only depends on the size of CNF, but also depends on the graph size. The CNF size for benchmark jsoup is smaller than dyn4j, however, the situation of graph size is the other way around, and the solving time for these two projects are very close.

- *Table 5.4: Inference for Dataflow Type System by LogiQL back end*

  The number of entity, functional predicate, and declaring derivation rules depends on the type system as we discussed in 3.3.1, for Dataflow Type System, the number is 41.

Similar to the Max-SAT encoding, the more constraint we get, the more logiQL data will be generated.

- *Comparison between Table 5.3 and Table 5.4:*

  If we compare the overall timing between Max-SAT and LogiQL back end, we noticed that the Max-SAT back end is faster. The most of the time consumed by LogiQL back end is when we add new data to the database, the database needs to spend some time to calculate the values in different entities.

  The serialization process for Lingeling solver and LogiQL back end are very similar to each other, and the serialized result is in string format. So we can notice that the serialization time for these two are close.

- *Table 5.5: Inference for OsTrusted Type System by Max-SAT back end*

  The pattern of this timing statistics are same as Table 5.3, but since the used graph separation algorithm is different, the size of the graph is much smaller than Dataflow Type System, and the corresponding timing data is also smaller than what we have in Table 5.3.

- *Comparison between Table 5.5 and Table 5.6:*

  From these two tables, we observed that solving the constraints as a whole is much faster than solving them separately. So in our case the overhead of multi-threading/process and the graph separation approach is too large. This can also explain the reason that the solving time depends on both CNF size and graph size, since the system will create a thread for each component. So from this comparison, we can have a conclusion that for the type system, for which the constraint separation is not mandatory, solving the constraint without the constraint separation is better for the performance.

- *Comparison between Table 5.7 and Table 5.8:*

  We can get the same conclusion that we got from the comparison between Table 5.5 and Table 5.6, the constraint separation has high overhead.

After the experimentation and the analysis, we have the conclusion that the graph separation has high overhead. For Dataflow Type System, the graph separation is a mandatory step for the type inference process, so we will have to do this pre-inference step. However, for the type system that all constraints can be solved as a whole, it is good for performance if we don't separate the constraint into different sets. For LogicBlox and Lingeling, one

Benchmark	Graph Size	CNF Size		Time (Millisecond)										
		Variable	Clause	Graph Generation	Sat4j					Lingeling				
					Parallel			Sequential		Parallel			Sequential	
					Sum of Serial-ization Time of All Threads	Sum of Solving Time of All Threads	Overall Process-ing Time of All Threads	Serialization Time	Solving Time	Sum of Serial-ization Time of All Threads	Sum of Solving Time of All Threads	Overall Process-ing Time of All Threads	Serialization Time	Solving Time
ant-javacard	37	556	1676	10	53	18379	1048	3	18029	17	3365	194	2	153
jdeb	160	94468	357415	235	2161	81238	3155	172	79666	1184	15553	1050	27	1138
jReactPhysics3D	113	452384	2194431	833	18837	68321	4590	134	137488	16799	16739	1241	107	2464
jsoup	285	453488	2429942	1333	21619	132964	6306	84	237892	26098	24019	1770	35	4000
dyn4j	235	1477558	6676218	3308	38272	155492	8854	96	235116	46985	33463	3529	306	7400
ode4j	402	6283428	29230245	13887	506342	503316	52611	925	403378	321177	105136	18612	893	25694

Graph Size is the number of component (set of constraint) after constraint separation. CNF Size gives the number of boolean variables and clauses in the CNF encoding. Timing consists of two parts: the timing of Sat4j solver and lingeling solver. In each part, there are timing results with both parallel and sequential solving options. For solving in sequential, Serialization Time and Solving Time are the sum of serialization time and solving time (the time for solving the clauses by SAT solver) of each component separately. For solving in parallel, for example, in following axis, if there are two threads t_1 and t_2, SE_S and SE_E stands for serialization starts and ends, and SO_S and SO_E stands for solving starts and ends. Sum of Serialization Time of All Thread would be $(t_1SE_E - t_1SE_S) + (t_2SE_E - t_2SE_S)$. Sum of Solving Time of All Threads would be $(t_1SO_E - t_1SO_S) + (t_2SO_E - t_2SO_S)$. Overall Processing Time of All Threads would be $(t_1SO_E - t_1SE_S)$

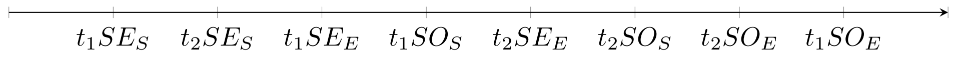

Table 5.3: Timing Result of Inferring Benchmarks with Dataflow Type System by Max-SAT Back End

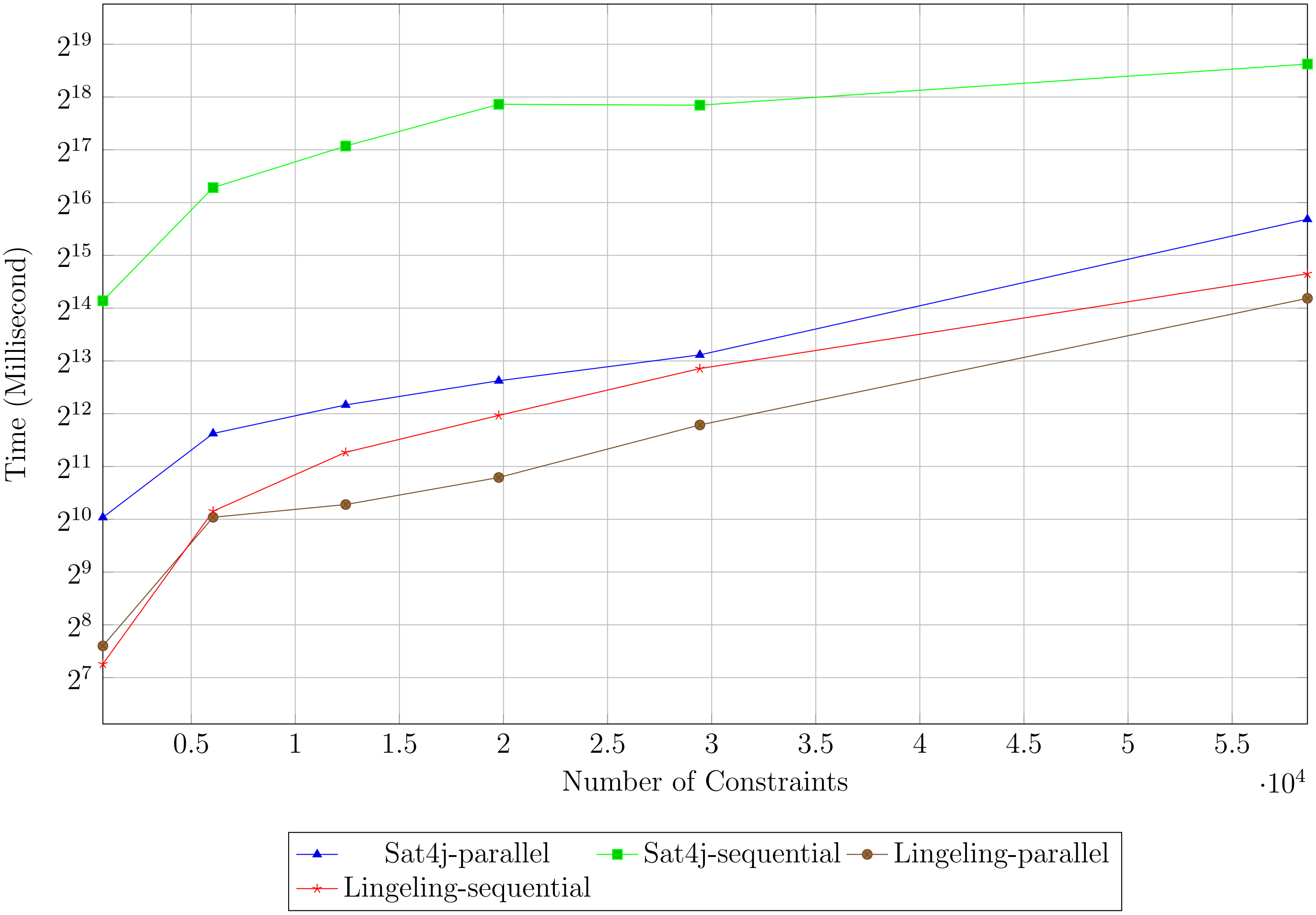

Figure 5.1: Relationship between Size of Constraint and Solving Time in Table 5.3

Benchmark	LogiQL Size		Time (Millisecond)		
	Predicates	Data	Graph Generation	Serialization Time	Solving Time
ant-javacard	41	844	10	13	65951
jdeb	41	8014	126	72	312744
jReactPhysics3D	41	19997	200	102	348621
jsoup	41	26348	422	208	550789
dyn4j	41	34555	606	183	606237
ode4j	41	88160	1094	504	1261515

Predicates gives the number of entity, functional predicate, and declaring derivation rules. As we explained in 3.3.1, for different type systems, the entity encoding are same, and the encoding for functional predicate and declaring derivation rules also have very similar logic. Data gives the number of facts that are added to the LogicBlox by delta modifier. The time related numbers have same meaning as the sequential approach in Table 5.3. For example, if we have two sets of constraint c_1 and c_2 need to be solved, SE_S and SE_E stands for serialization starts and ends, and SO_S and SO_E stands for solving starts and ends. The timeline of processing the sets of constraint will follow below axis. The Serialization Time is $(c_1SE_E - c_1SE_S) + (c_2SE_E - c_2SE_S)$, and the Solving Time is $(c_1SO_E - c_1SO_S) + (c_2SO_E - c_2SO_S)$.

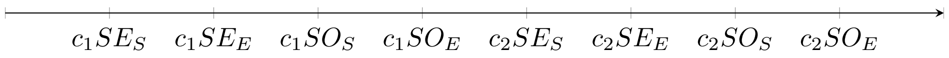

Table 5.4: Timing Result of Inferring Benchmarks with Dataflow Type System by LogiQL Back End

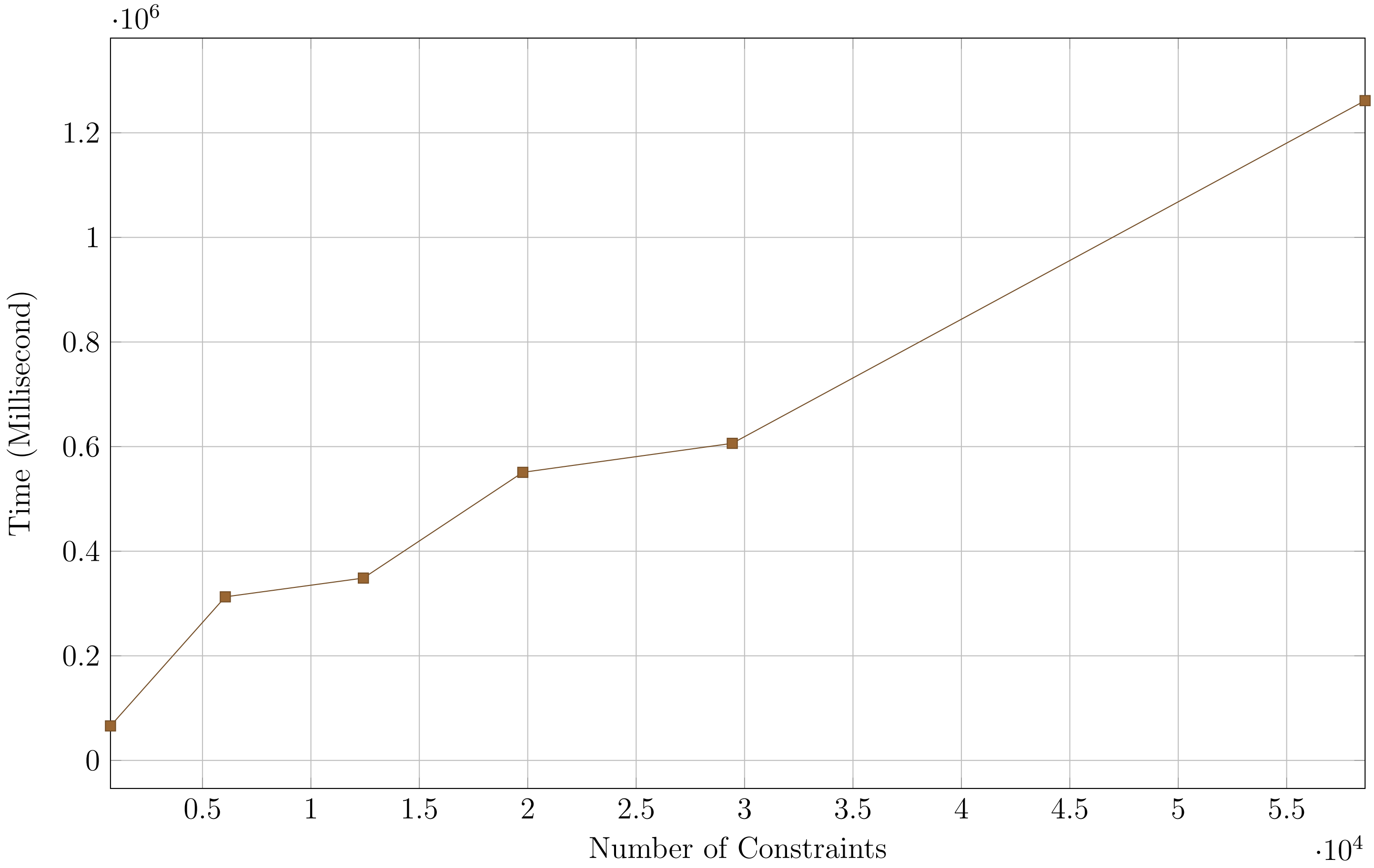

Figure 5.2: Relationship between Size of Constraint and Solving Time in Table 5.4

Benchmark	Graph Size	CNF Size		Time (Millisecond)										
		Variable	Clause	Graph Generation	Sat4j					Lingeling				
					Parallel			Sequential		Parallel			Sequential	
					Sum of Serialization Time of All Threads	Sum of Solving Time of All Threads	Overall Processing Time of All Threads	Serialization Time	Solving Time	Sum of Serialization Time of All Threads	Sum of Solving Time of All Threads	Overall Processing Time of All Threads	Serialization Time	Solving Time
ant-javacard	4	1209	4224	32	24	2004	561	10	2004	13	295	94	9	105
jdeb	11	9879	36845	354	90	5598	650	29	5507	49	1674	173	31	235
jReactPhysics3D	36	21708	86129	690	171	19575	1102	63	17596	161	10417	567	69	520
jsoup	31	27567	130032	2310	278	17598	1095	73	15526	154	10705	453	94	507
dyn4j	53	45978	214238	3614	372	35794	1640	99	26553	177	21300	794	93	771
ode4j	204	92331	439142	7527	770	265315	9681	221	201391	780	85507	2990	182	4700

The meaning of each column is same as Table 5.3.

Table 5.5: Timing Result of Inferring Benchmarks with OsTrusted Type System by Max-SAT Back End with Constraint Separation

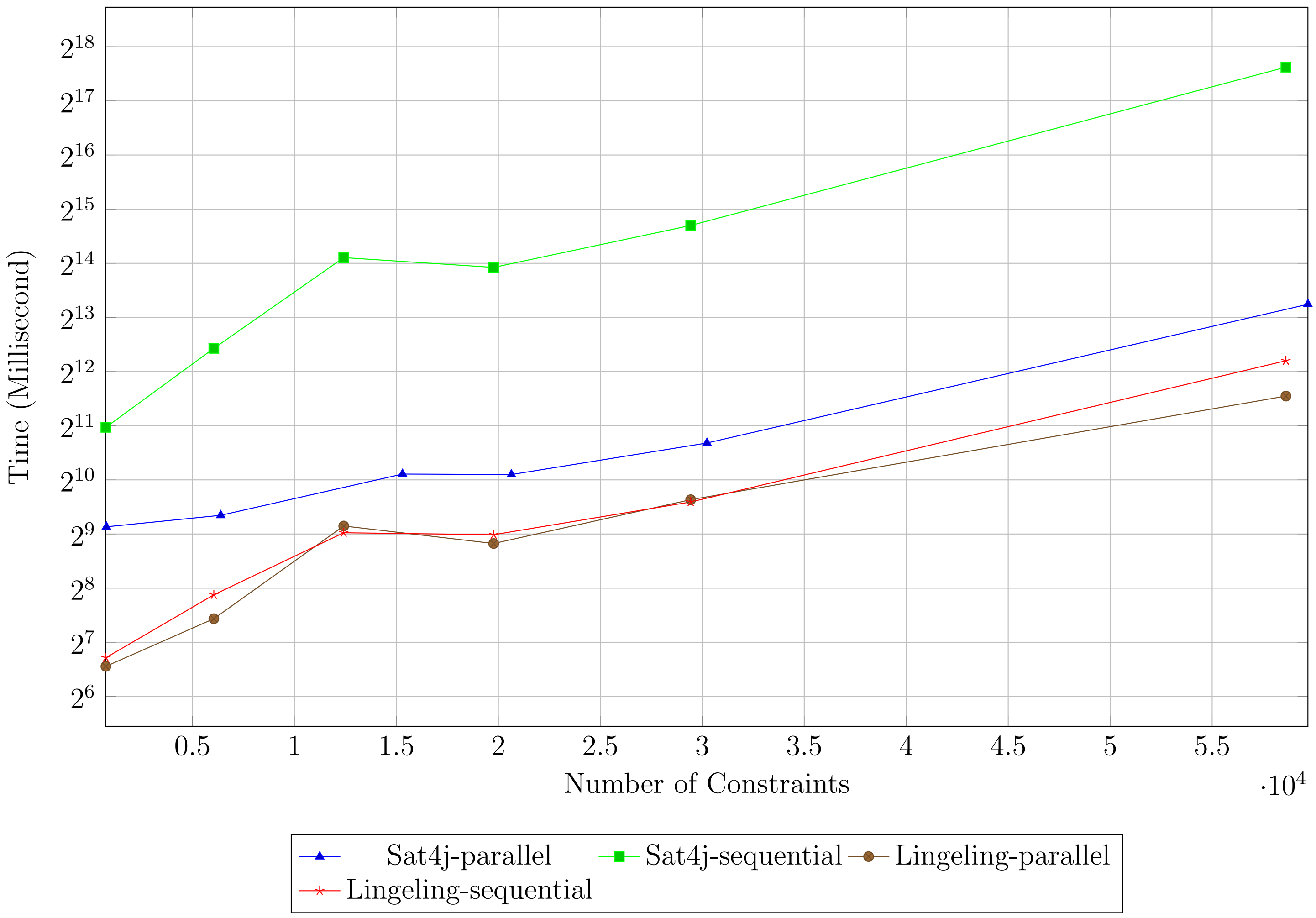

Figure 5.3: Relationship between Size of Constraint and Solving Time in Table 5.5

Benchmark	Time (Millisecond)			
	Sat4j		Lingeling	
	Serialization Time	Solving Time	Serialization Time	Solving Time
ant-javacard	8	500	11	99
jdeb	35	639	28	286
jReactPhysics3D	65	749	60	689
jsoup	83	832	75	693
dyn4j	121	870	107	568
ode4j	171	1136	254	1351

The meaning of each column is same as Table 5.5, and because in this case we process all the constraints as a whole, the Serialization Time and Solving Time are the time duration between the start and the end of the corresponding steps. The CNF data also keeps same as the previous table.

Table 5.6: Timing Result of Inferring Benchmarks with OsTrusted Type System by Max-SAT Back End without Constraint Separation

Benchmark	LogiQL Size		Time (Millisecond)		
	Predicates	Data	Graph Generation	Serialization Time	Solving Time
ant-javacard	57	1119	29	3	14635
jdeb	57	9705	320	14	87422
jReactPhysics3D	57	22732	670	41	218423
jsoup	57	30457	2344	52	300643
dyn4j	57	45470	2376	61	552948
ode4j	57	91057	3349	123	2026301

The meaning of each column is same as Table 5.7. The LogiQL data also keeps same.

Table 5.7: Timing Result of Inferring Benchmarks with OsTrusted Type System by LogiQL Back End with Constraint Separation

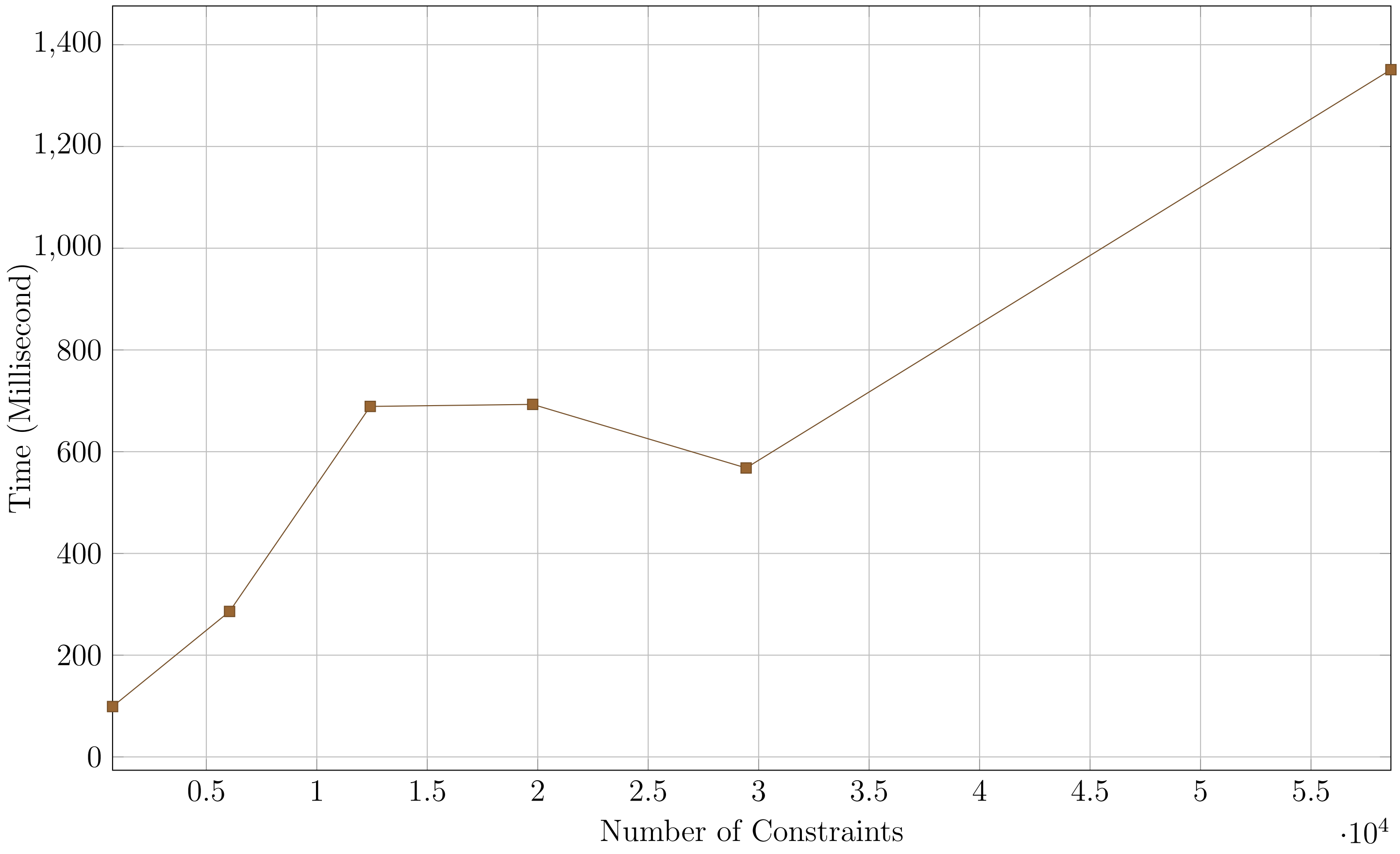

Figure 5.4: Relationship between Size of Constraint and Solving Time in Table 5.6

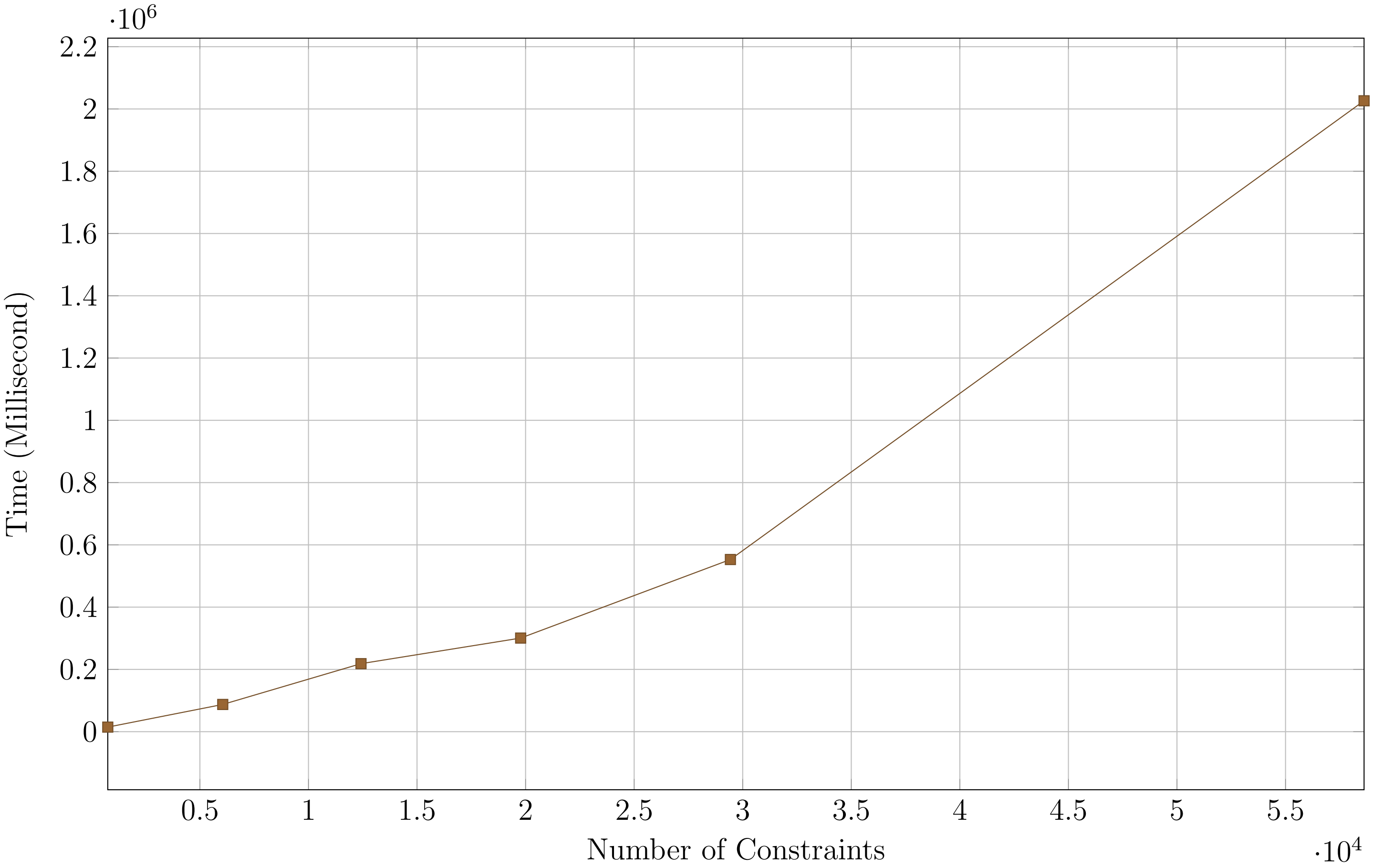

Figure 5.5: Relationship between Size of Constraint and Solving Time in Table 5.7

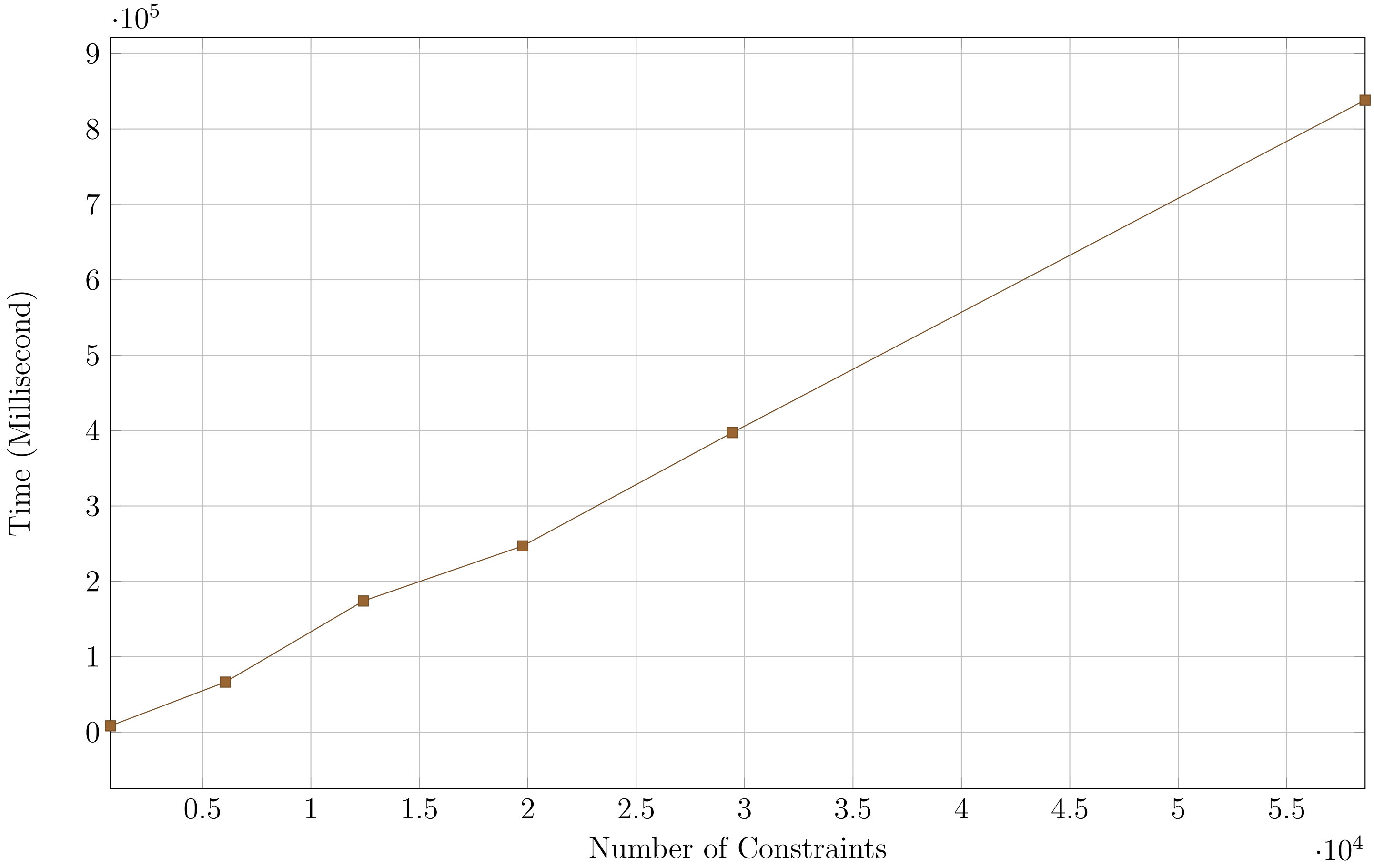

Figure 5.6: Relationship between Size of Constraint and Solving Time in Table 5.8

Benchmark	Time (Millisecond)	
	Serialization Time	Solving Time
ant-javacard	4	8398
jdeb	13	66407
jReactPhysics3D	32	174018
jsoup	34	247030
dyn4j	43	397290
ode4j	89	838210

The meaning of each column is same as Table 5.7. The LogiQL data also keeps same.

Table 5.8: Timing Result of Inferring Benchmarks with OsTrusted Type System by LogiQL Back End without Constraint Separation

possible overhead is the I/O operation. Our tool will need to generate a textual file and then call LogicBlox or Lingeling to read it. If there are too many files generated, it's time consuming to keep reading these files over and over. Another overhead is when we separate the constraints, one constraint may be present in multiple sets, for example, in Figure 4.2, the subtype constraint between $obj1$ and $obj2$ will be present in both string component and integer component. So we will solve that constraint twice in that case, which would spend more time.

By comparing the timing result of parallel and sequential approach, we found out the parallelization is good for performance in general. However, during the experimentation, we ran into GC overhead limit exceeded exception when there are too many (around 5000) threads. If we choose the sequential approach, the overall time would be very long, but we got the result eventually. So if we separate the constraint into thousands of components, we need to consider about the trade off between performance and successful inference.

The relationship between the growth of the overall consumed time and the size of the project is in positive correlation, which aligns with our expectation. But as I mentioned above, the overall timing also depends on the size of graph. For Dataflow Type System, the constraint size of project dyn4j is larger than jsoup by 30%, but there is no much difference between the overall solving time of these two projects. The reason of this result is that Jsoup has 50 more components than dyn4j after the graph separation, and according to the analysis of two previous paragraphs, the more components would use more time process, so that it slows the overall process down.

The used hardware was a remote server with 4 sockets and 8 cores per socket. Each CPU is GenuineIntel at 3301 MHZ running Ubuntu 14.04 Linux 64 bit and using Oracle JDK 8. The total free memory is 42 GB, but the Java heap space is limited to 5 GB.

Chapter 6

Future Work

6.1 Future Work on Type Constraint Solver

The Type Constraint Solver provides a system that can solve type constraint by SAT solver or LogicBlox. However, the LogiQL encoding is inspired by Max-SAT encoding, and both encodings have similar concept of interpretation for type constraint. There are lots of features in LogiQL language and LogicBlox haven't been used, so we plan to explore the full potential of the LogicBlox database, such that we can make more optimizations and improvements for the LogiQL encoding. One possible improvement is as I mentioned in Section 3.3.1, we haven't found a way to encode the preference constraint into the LogiQL form. So if Checker Framework Inference generates preference constraint for the given program, it won't be solved by LogiQL back end. We are going to do more research on LogicBlox, and we believe there are some ways to encode preference constraint into LogiQL. If we have multiple ways to encode type constraint in LogiQL, we also would like to compare the performance among different encodings.

At this moment, we did lots of testing on type systems with two type qualifiers, Dataflow Type System and OsTrusted Type System, and they both integrated with Type Constraint Solver well. However, in the encoding process, they are both treated as two type qualifiers type system. Type Constraint Solver is designed for solving type constraints from any type systems, so one future work is run Type Constraint Solver with more complex type systems. One type system we are focusing on is Generic Universe Type System[18]. We have been working on integrating it into our solver, and then we can compare the inferred result between Type Constraint Solver and the solver mentioned in the paper.

Although in our experimentation, the performance of Max-SAT solver with Lingeling solver looks good, the largest benchmark ode4j is not large enough to show the system's performance. And the current back ends are still not good for interactive usage. We believe there are more forms that can be used to solve the type constraints. Since Type Constraint Solver is easliy extended with more back ends, we would love to develop more back ends into our system.

We would like to make the system scalable enough so that can handle larger benchmarks. With current Checker Framework and Checker Framework Inference, ode4j is the largest project that can be successfully inferred. Checker Framework Inference crashed when we tried bigger projects. We will fix the bugs and optimize both frameworks, and make the whole system more reliable.

More statistics would provide us more information about Type Constraint Solver's performance, so we will try inferring more big projects, and find out the limitation of the system.

6.2 Future Work on Dataflow Type System

Dataflow Type System provides us all possible run time types for a given location. In order to verify the correctness of the inferred result, we plan to find some existing tools that can do similar analysis, and compare the result from existing tools and the Dataflow Type System.

During our development, we keep running into problems when the test case becomes complicated. We fixed lots of bugs that can be triggered by some corner cases. For Dataflow Type System, we will try it on more test cases, fix the occurred bugs, and make the system robust.

In Section 2.4, we mentioned that *Constant Value Checker* can infer variable's value and type in compile for type boolean, integer, double and string. So we would like to make *Dataflow Type System* also infers values. One possible approach is we can let Dataflow type qualifier take other parameters for values. Then for the literal base case, we let the checker not only record the type but also fill parameter with the values. Like what we did for the parameter *typeNames*, for least upper bound of the value parts of two type qualifiers, we will merge the values from two qualifiers into one set.

Chapter 7

Conclusion

In this thesis, we presented a system, *Type Constraint Solver*, that can solve the type constraint generated from *Checker Framework Inference* for an arbitrary inferable type system. *Type Constraint Solver* provides two encoding strategies that can encode the type constraint into Boolean formulas or LogiQL language form, use existing Max-SAT solvers and LogicBlox to solve the encoding accordingly, and get the exact type qualifier for each slot. *Type Constraint Solver* also provides two constraint separation strategies that can separate constraints into groups, and the options to solve the groups in parallel or sequential. The infrastructure of *Type Constraint Solver* is designed to be easily extended with custom encoding logic and even new back ends for further development.

We also introduced an inferable type system *Dataflow Type System*, which is built on the top of *Checker Framework Inference*. The type system can infer all possible run time Java types of method, field, variable, and parameter in compile time. We took the advantage of the scalability of *Type Constraint Solver* and built a custom constraint solving approach for *Dataflow Type System* on the top of it.

For experimentation, we applied our tools to six real-world, open source Java projects. We run *Type Constraint Solver* with *Dataflow Type System* and *OsTrusted Type System* on these projects with different back ends and solving options. Through analysis of the experimentation statistics, we explained the trade off of the constraint separation and the benefit of parallelization constraint solving.

References

[1] Checker Framework Inference. https://github.com/typetools/checker-framework-inference. Accessed February 01 2017.

[2] CNF files format. http://www.satcompetition.org/2009/format-benchmarks2009.html. Accessed January 02 2017.

[3] Connectivity (graph theory). https://en.wikipedia.org/wiki/Connectivity_%28graph_theory%29. Accessed July 29 2016.

[4] Constant Value Checker. https://checkerframework.org/manual/#constant-value-checker. Accessed March 01 2017.

[5] Lingeling SAT Solver. http://fmv.jku.at/lingeling/. Accessed July 29 2016.

[6] Maximum Satisfiability problem. https://en.wikipedia.org/wiki/Maximum_satisfiability_problem/. Accessed May 31 2016.

[7] The Checker Framework. http://types.cs.washington.edu/checker-framework/. Accessed July 29 2016.

[8] Type Annotations and Pluggable Type Systems. https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. Accessed July 29 2016.

[9] Type annotations specification (JSR308). https://checkerframework.org/jsr308/. Accessed February 01 2017.

[10] Where is Java used in Real World? http://javarevisited.blogspot.ca/2014/12/where-does-java-used-in-real-world.html. Accessed February 01 2017.

[11] Rahul Agarwal and Scott D. Stoller. *Type Inference for Parameterized Race-Free Java*, pages 149–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[12] A. Aiken and E. L. Wimmers. Solving systems of set constraints. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Jun 1992.

[13] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[14] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 57–74, New York, NY, USA, 2006. ACM.

[15] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[16] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 56–69, New York, NY, USA, 2001. ACM.

[17] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, July.

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

[19] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[20] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[21] Cormac Flanagan and Stephen N. Freund. Type inference against races. *Sci. Comput. Program.*, 64(1):140–165, January 2007.

[22] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 321–336, New York, NY, USA, 2007. ACM.

[23] Terry Halpin and Spencer Rugaber. In *LogiQL*. CRC Press, Boca Raton, 2015.

[24] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 207–218, New York, NY, USA, 2004. ACM.

[25] M W Krentel. The complexity of optimization problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 69–76, New York, NY, USA, 1986. ACM.

[26] Ondřej Lhoták and Laurie Hendren. *Scaling Java Points-to Analysis Using Spark*, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[27] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[28] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013.