Complexity Analysis of Tunable Static Inference for Generic Universe Types

by

Nahid Juma

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, 2015

Author's Declaration

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

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

Abstract

This work studies the computational complexity of a tunable static type inference problem which was introduced in prior research [1]. The problem was assumed to be inherently difficult, without evidence, and a SAT solver was used to obtain a solution. In this thesis, we analyze the complexity of the inference problem. We prove that it is indeed highly unlikely that the problem can be solved efficiently. We also prove that the problem cannot be approximated efficiently to within a certain factor. We discuss the computational complexity of three restricted but useful versions of the problem, showing that whilst one of them can be solved in polynomial time, the other two are still inherently difficult. We discuss our efforts and the roadblocks we faced while attempting to conduct experiments to gain further insight into the properties which distinguish between hard and easy instances of the problem.

Acknowledgements

I am thankful to my parents and to Shaneabbas for always being supportive of my aspirations. I will forever be grateful for my son, Mahdi, who is a constant source of joy in my life.

I would like to thank my supervisors, Professor Mahesh Tripunitara and Professor Werner Dietl, for guiding and sharing their valuable insights with me throughout this research.

Table of Contents

List of Figures

Chapter 1

Introduction

Software is increasingly complex. Programming languages such as Java, with which software is created, provide the notion of a 'type' so programmers are able to classify data, and enforce associated rules on how data may flow when the software runs. Typing is useful to analyse, and sometimes guarantee, properties - for example, whether every state that the software reaches is safe.

It is recognized widely that the type system that comes built-in with languages such as Java is inadequate. While it does help us prevent some errors, many others escape it - for example, errors caused by multiple references to the same object. To promote error-free code, several languages support pluggable type systems. These are optional type systems designed to detect specific errors which may otherwise be overlooked.

An example is the Generic Universe Type (GUT) system [2], an ownership-based type system which is integrated into the tool suite of the Java Modelling Language. Such a system organizes the objects in a program in a hierarchical structure where some objects own others, and enforces the rule that objects can be modified only by their owner and peers. While this helps to prevent several errors, a major drawback of this system is that it requires a significant amount of code annotation. Having to manually annotate code is a burden for the programmer. It is therefore preferable to have algorithms that automatically infer the annotations.

In ownership type systems, there is no single most general typing, rather a program may have multiple valid typings. The desired typing is the one which describes the ownership topology preferred by the programmer. In order for the inference to reflect the programmer's intentions, prior work [1] has proposed the idea of 'tuning' the type inference for GUT. In addition to the source code, the inference tool is given optional

ownership assignments with weights. The inference problem is to find the assignment of types to objects that satisfies all the constraints imposed by the type system and yields the maximum weight. In [1], this problem is implicitly assumed to be inherently difficult. The approach adopted to solve it is to reduce it to a boolean satisfiability problem and then employ a partial weighted Max-SAT solver. In this thesis, we present a detailed analysis of the problem's computational complexity.

1.1 Motivation

In computer science, and related fields, when attempting to solve a computational problem, a natural and fundamental question to pose is: how hard is the problem? The answer to this question helps to decide what an appropriate solution approach to the problem is. This question was not addressed in [1] and in this thesis we attempt to fill this gap.

We undertook this work because we felt that identifying which computational complexity class the tunable type inference problem falls under would prove useful in several ways. If the problem was found to be tractable, we would be able to construct a polynomial time algorithm to solve every possible instance of it. In this case, the algorithm would then be evaluated against the performance of the SAT solver approach. Since static inference occurs at compile time, from the programmer's standpoint, efficient inference algorithms are preferred. Furthermore, such a polynomial time algorithm could then potentially be used to efficiently solve the tunable type inference problem for other type systems that are similar to the GUT system.

If, on the other hand, the problem was found to be intractable, this result would also be useful. Firstly, it would provide a formal justification for the solution approach taken in [1]. Secondly, it would confirm the existence of hard instances for which the inference tool may take exponential time to solve. This insight is crucial in order to be able to give any kind of guarantee about the tool's performance. In this case, we would then explore the sources of intractability and investigate the possibility of mitigating them. We would also try to identify the characteristics of hard instances of the problem and explain why all the benchmarks used in [1] appear to be tractable instances.

In summary, this research will advance knowledge in software, particularly in the context of type systems. Efficient inference can potentially reduce compilation time. This will encourage programmers to adopt ownership type systems which in turn will result in increased reliability of software. In addition the insight gained can be extended to other inference problems of a similar nature.

1.2 Problem Statements

We formally pose the questions that this thesis attempts to answer as follows:

> Is the tunable type inference problem for GUT tractable? That is, does there exist a polynomial-time algorithm which solves it?
>
> If it is intractable and cannot be solved efficiently exactly, can it at least be approximated efficiently?
>
> Are there any restricted versions of the problem which are useful and tractable?
>
> What are some of the sources of intractability? Why is it that all the benchmarks used in [1] appear to be tractable? What does a hard instance look like?

1.3 Related Work

The computational complexity of various other type inference problems has been subjected to prior research. Flanagan et al. [4] proved that the inference problem for a race condition checking type system, `rccjava`, is **NP**-complete. They then used this result to motivate their approach to solving the inference problem by a reduction to propositional satisfiability.

Elder et al. [5] proved that it is unlikely that a polynomial time algorithm exists to solve the dynamic heap type inference problem. This is the problem of inferring program-defined types for each allocated storage location in the heap, and identifying "untypable" blocks which reveal heap corruption or type safety violations.

Aldrich et al. [6] introduced AliasJava, an annotation system for Java that makes alias patterns explicit in the source code, and give an algorithm for automatically inferring the alias annotations. Their system involves equality, component and instantiation constraints. The type system we deal with involves different constraints such as viewpoint adaptation and subtyping.

Chin et al. [7] propose CLARITY for the inference of user-defined qualifiers for C programs based on user-defined rules, which can also be inferred given user-defined invariants. CLARITY uses a graph-based propagation algorithm to infer several type qualifiers, including `pos` and `neg` for integers, `nonnull` for pointers, and `tainted` and `untainted` for strings. These type qualifiers are not context-sensitive. We focus on a type inference for Java which is context-sensitive i.e. ownership-based.

A work that is closely related to ours is that of Huang et al. [8]. They devise a polynomial time algorithm to solve type inference for the Universe Type system, and therefore implicitly prove that the problem is in **P**. Universe Types is an ownership type system closely related to Generic Universe Types; the former is not able to handle generics whilst the latter is. Our work is different from [8] in two ways. Firstly, in order to handle generics GUT allows equality constraints in which both sides are constraint variables. It can be verified that the Set Based Solution they propose is able to handle this additional kind of constraint. Secondly, we are interested in a type inference which is tunable, that is, guided by the programmer's preferences, whereas the solution proposed in [8] can only infer the deepest ownership structure.

To our knowledge, there has been no prior work that has analysed the computational complexity of tunable static inference for an ownership type system such as GUT.

1.4 Outline

The remainder of this work is organized as follows.

In Chapter 2 we briefly explain the computational complexity concepts used in this thesis.

In Chapter 3 we provide a background on the Generic Universe Type system. We explain the tunable inference problem for GUT as well as the approach taken in [1] to solve it. We also summarize their implementation experience. The majority of the contents of this chapter has been borrowed and reworded from [1]. However, what [1] refers to as a 'boolean encoding', we formally present as a reduction from a GUT tunable inference instance to a partial weighted Max-SAT instance and we prove the soundness of this reduction.

In Chapter 4, we present our computational complexity analysis. We prove that the tunable type inference (TTI) problem for the GUT system is **NP**-complete. In other words, that it is highly unlikely that there exists a polynomial time algorithm which solves

the problem exactly. We then establish an inapproxamability result which proves that the problem is also difficult to approximate to within a certain factor. The complexity of three restricted but practical versions of the problem are discussed. We discuss our thoughts on how to identify the sources of complexity and the structural properties which differentiate between hard and easy instances of the problem.

Finally, in Chapter 5, we summarize the conclusions of this research and discuss the opportunities for future work.

Chapter 2

Background on Computational Complexity

A *computational problem* is a general question to be answered, usually possessing several parameters, whose values are left unspecified. A problem is described by giving a general description of all its parameters and a statement of what properties the answer, or *solution*, is required to satisfy. An *instance* of the problem is obtained by specifying particular values for all the problem parameters. As an example, consider the tunable type inference problem which is the focus of this thesis. This is a computational optimization problem. The parameters of this problem are a compilable program, P, which can be partially annotated with types, and a set of optional, weighted constraints B. A solution is the assignment to the constraint variables which makes the program type check and which achieves the maximum possible weight from the breakable constraints. If there is no assignment which makes the program type check then the output should be 'no solution' or similar.

An *algorithm* is a step-by-step procedure for solving a problem. Without any loss of generality, an algorithm can be considered to be a computer program written in a precise computer language. An algorithm is said to solve a problem if it can be applied to any instance of the problem and is guaranteed to produce the solution for that instance. There could be multiple algorithms which solve a problem, but in general, we are interested in the most *efficient* one. In its broadest sense, the notion of efficiency, refers to all the various computing resources needed to execute an algorithm. However, time requirements are often the most dominant factor, and by the 'most efficient algorithm' one normally means the fastest one.

The description of a problem instance that we provide as input to the computer can be

viewed as a single finite string of symbols chosen from a finite input alphabet. Although there are many ways to map instances to strings, for simplicity and without any loss of generality, let us assume that each problem has a fixed encoding scheme, which maps its instances into strings describing them. The *input length* of an instance of a problem is used as a formal measure of the size of the instance, and is defined to be the number of symbols, n, in the description of the instance obtained from the problem's encoding scheme. It is reasonable to expect that the relative difficulty of problem instances vary roughly with their sizes. Hence the time requirements of an algorithm can be conveniently represented in terms of n. The *time complexity function* of an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size.

The *computational complexity* of a problem refers to the inherent difficulty of the problem. A problem is inherently difficult if it requires an exponential amount of time to be solved, irrespective of the algorithm which is used to solve it. Problems are classified based on their inherent difficulty with certain relationships existing between classes. One of the central objects of study in computational complexity theory are decision problems because they are easier to analyze than the other types of problems such as optimization, counting and search problems. In a decision problem, given an input instance, the output is a yes/no answer which verifies if the input satisfies a certain property. For example, a suitable decision version of the tunable type inference optimization problem is as follows:

TTI = $\{\langle P, B, k\rangle : \exists$ an assignment to the constraint variables of program P that makes P type check and achieves a weight of at least k from the breakable constraints $B\}$

In the above, k, is a positive integer. If a TTI instance x satisfies the above property we say that x is a 'yes' instance and that $x \in TTI$. Conversely, if x does not satisfy the property, we say that x is a 'no' instance and that $x \notin TTI$. It turns out that if one can solve this decision version of the tunable type inference problem, then one can solve the optimization version with only a polynomial amount of extra overhead. In other words, the optimization version is no harder than the decision version of the problem. To see this, assume we have an algorithm to solve the TTI decision problem. Given the TTI optimization problem, we can use this algorithm in conjunction with a binary search to find the optimal assignment. A binary search is highly efficient with a worst case running time of $O(log(n))$.

We now define the complexity classes which are of interest to us. A decision problem is said to be in class **P** if it can be solved in polynomial time. That is, there exists an algorithm to solve it, whose time complexity function is $O(p(n))$ for some polynomial function p. Such a problem is efficiently solvable. A decision problem is said to be

in class **NP** if given a suitable certificate, the 'yes' instances of the problem can be verified in polynomial time. It follows that $\mathbf{P} \in \mathbf{NP}$ since if a problem can be solved in polynomial time, we can ignore the certificate and solve the problem to verify a 'yes' instance. Whether or not $\mathbf{P} = \mathbf{NP}$ is an open question. It is generally conjectured that $\mathbf{P} \neq \mathbf{NP}$. In order to define the other two classes of interest, we must introduce the notion of a reduction. Let A and B be two decision problems. We say that A reduces to B, denoted $A \leq B$, if there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. Two immediate observations are:

1. If $A \leq B$ and $B \in P$, then also $A \in \mathbf{P}$ (conversely, if $A \leq B$, and $A \notin \mathbf{P}$ then also $B \notin \mathbf{P}$)
2. If $A \leq B$ and $B \leq C$, then also $A \leq C$.

A decision problem is said to be in class **NP**-hard if every problem in **NP** can be reduced to it. That is, a problem is in **NP**-hard if it is at least as hard as the hardest problems in **NP**. A decision problem is said to be in class **NP**-complete if it is both in **NP** and in **NP**-hard. A well known **NP**-complete problem is the boolean satisfiability problem, or SAT, in which the input instance is a boolean formula and the output is an assignment to the boolean variables which makes the formula evaluate to true, if such an assignment exists. It is highly unlikely that a problem that is **NP**-complete has an efficient solution. That is, they are inherently difficult. In the unlikely event that an efficient solution is found for any **NP**-complete problem, it would imply that efficient solutions to all **NP**-complete problems exist, and that $\mathbf{P} = \mathbf{NP}$. Since all **NP**-complete problems are mutually reducible to each other, computer scientists have focused on writing good solvers for SAT which can then potentially be used to solve all other **NP**-complete problems. Of course, even the fastest SAT solvers are still exponential on their worst-case inputs.

For ease of analysis, in our complexity analysis, we focus on the decision version of the tunable type inference problem.

Chapter 3

Background on Tunable Type Inference for GUT

In this chapter we describe the Generic Universe Type system. We explain the tunable type inference approach taken in [1], and we summarize and comment on their implementation experience. The majority of the contents of this chapter has been reworded from [1]. In this thesis however, we formalize their 'encoding' from tunable type inference to partial weighted Max-SAT as a reduction and prove that the reduction is sound.

3.1 Generic Universe Types

In GUT programmers are able to organize the objects in the heap in a hierarchical ownership topology. Each object is owned by at most one other object, its owner. The ownership relation is acyclic, resulting in a topology consisting of a forest of ownership trees, where objects without owners are roots. The set of objects sharing an owner is called a *context*. Modifications across context boundaries are restricted: an object can only be modified by its owner or by other objects within the same context as itself.

Ownership Modifiers A programmer describes the ownership topology by annotating each reference type in the source code with one of three possible ownership modifiers. An *ownership modifier* is an annotation given to the reference type which expresses its ownership relation to the current receiver object `this`. The three modifiers composing the surface syntax are:

```
public class Person {
  peer Person spouse;
  rep Account savings;
  rep List<peer Person> friends;

  int assets() {
    any Account a = spouse.savings;
    return savings.balance + a.balance;
  }
}
```

Figure 3.1. Example Program, a figure taken from [1].

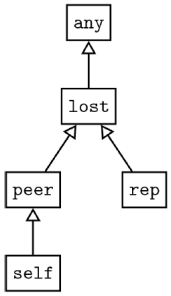

Figure 3.2. Ownership Modifier Type Hierarchy, a figure taken from [1].

- `peer` conveys that the referenced object is in the same context as the current object `this`. For example, in Figure 2.1, a `Person p` has the same owner as `p.spouse`.
- `rep` conveys that the referenced object is owned by the current object `this`. For example, in Figure 2.1, a `Person p` is the owner of `p.savings`.
- `any` gives no static information about the relationship between the two objects.

  In addition, GUT uses two internal ownership modifiers, which do not appear as explicit annotations in the source code:
- `lost` conveys that although there exists a relationship between the two objects, it is not expressible by the modifiers `peer` and `rep`. For example, in Figure 2.1, `spouse.savings` is a nephew of `this`. GUT cannot express this relationship so it assigns `spouse.savings` a modifier of `lost`.
- `self` is used exclusively for the current receiver object `this`.

Fig. 3.2 shows the type hierarchy. A `self` annotated type is a subtype of the corresponding `peer` type because since `self` denotes the `this` object, it is obviously a peer of `this`. `peer` and `rep` annotated types are subtypes of the corresponding type with a `lost` modifier because the latter conveys less ownership information. An `any` annotated type is a supertype of all other versions.

Generic Types, Viewpoint Adaptation and Encapsulation The example in Fig. 3.1 shows how the ownership modifiers are used with generic types. The field `friends` has type `rep List<peer Person>`. This conveys that the `List` object is owned by the `Person` object containing the field, whereas the elements stored in the list are peers of that `Person` object. We observe here that the ownership modifier `<peer`

`Person>` is interpreted with respect to the `Person` object containing the field, and not the object of the generic type.

For a compound expression, the overall modifier is determined by combining the modifiers of the components. For example, consider a field access `tony.spouse` where `tony` is of type `rep Person`. We first traverse over a `rep` modifier and then a `peer` modifier. The overall modifier is the result of adapting `tony's spouse` modifier from the viewpoint of `tony`, to the viewpoint of `this`. This yields `rep` because the resulting object, `tony's spouse`, has the same owner as `tony`, which is `this`.

Sometimes, viewpoint adaptation results in a loss of static ownership information. For example, consider the compound expression `spouse.savings`. First, we traverse over a `peer` field and then a `rep` field, so the resulting object has a specific relationship to the receiver object `this` but the relationship cannot be expressed in the type system. To express this loss of information, GUT uses the modifier `lost`. Two different expressions of `lost` type might represent different unknown relations, hence it would be illegal to assign an object of `lost` type to another. GUT therefore prohibits the left-hand side of assignments from having a `lost` type. This is the reason why GUT introduces a `lost` type rather than reusing `any` since it would be too restrictive to disallow `any` on the left-hand side of assignments.

Viewpoint adaptation can be formulated as a function $\triangleright$ which takes two ownership modifiers and yields the adapted modifier. (1) $\texttt{peer} \triangleright \texttt{peer} = \texttt{peer}$; (2) $\texttt{rep} \triangleright \texttt{peer} = \texttt{rep}$; (3) $u \triangleright \texttt{any} = \texttt{any}$, (4) $\texttt{self} \triangleright u = u$; and (5) for all other combinations, the result is `lost`. Besides field accesses, viewpoint adaptation is also applicable to parameter passing, result passing, and type variable bound checks.

In GUT, the programmer has the option of enforcing an encapsulation scheme called *owner-as-modifier*. In this scheme, an object can be referenced by any other object, but can only be modified by a reference chain which passes through its owner. This permits owners to control state changes of their owned objects and thus maintain invariants.

3.2 Tunable Static Inference Approach

An unannotated Java program is legally-typed because the default modifier is `peer`. However, this default typing is usually undesirable because it describes a flat ownership structure which imposes no restrictions nor provides any guarantees about the program's behaviour. On the other hand, manually annotating the source code is a burden for the programmer. Therefore inference is needed to automatically produce annotations which

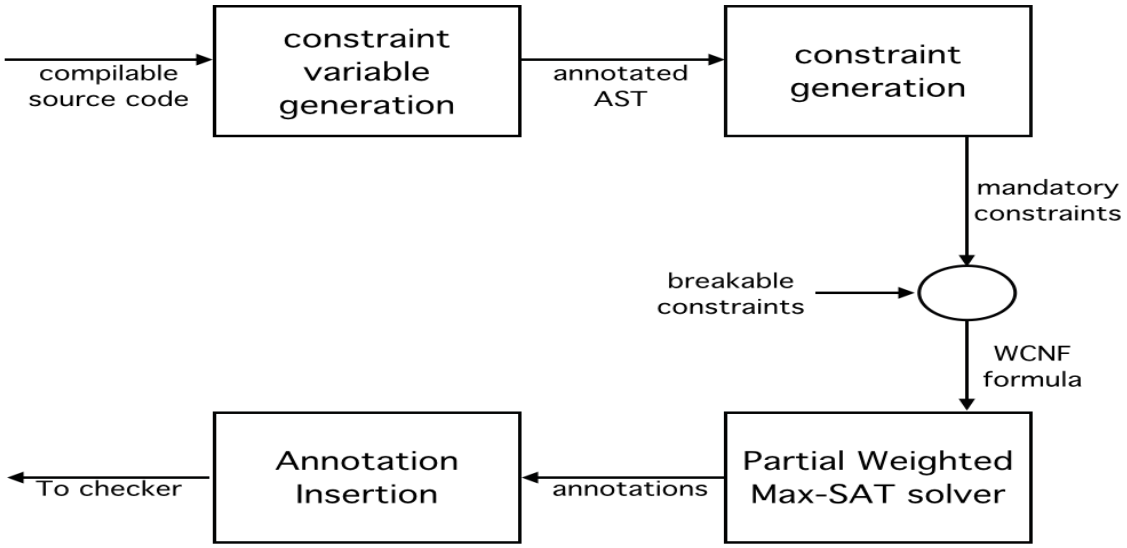

Figure 3.3. Overview of the inference approach.

describe ownership structures reflecting the programmer's design intentions.

Fig. 3.3 overviews the tunable type inference process. There are four primary steps: generating constraint variables, creating type constraints over these variables, obtaining breakable constraints if any, and solving the constraints to infer a typing.

Constraint Variables For each position in the source code where a concrete modifier may occur, a constraint variable α is introduced which represents the modifier for that position. The goal of the inference is to assign one of the modifiers `peer`, `rep`, or `any` to each of these constraint variables. Constraint variables are also generated for each expression which gives rise to viewpoint adaptation; these will be assigned from the modifiers `peer`, `rep`, `any`, or `lost`. Fig. 3.4 shows an example input program with constraint variable locations numbered. α_1 to α_9 correspond to locations where ownership modifiers may explicitly appear in the source code. α_{10} is the result of a viewpoint adaptation. It represents the result of adapting the declared upperbound of the type variable of class `List`, which is assumed to be `any Object`, from the viewpoint of `patients` to `this`. Hence $\alpha_{10} = \alpha_2 \triangleright$ `any`, which trivially evaluates to `any`.

Mandatory Constraints The inference then traverses the program's Abstract Syntax Tree and generates constraints over the constraint variables. Each programming construct

```
class Doctor{

     Doctor1 substitute;
     List2<Patient3>10 patients;

     void addPatient(Patient4 p){
          patients.add(p);
     }

     void addSubstitute(Doctor5 d){
          substitute = d;
     }

     void demo(){
          Doctor6 d1 = new Doctor7();
          Doctor8 d2 = new Doctor9();
          this.addSubstitute(d1);
     }
}
```

Figure 3.4. Example program with constraint variable locations numbered.

1. α_2 List $< \alpha_3$ Patient$> \triangleright$ any Object = α_{10}
   (the upperbound of the type variable of class `List` is assumed to be `any Object`)
2. α_3 Patient $<: \alpha_{10}$
3. α_4 Patient $<: \alpha_3$ Patient
4. α_5 Doctor $<: \alpha_1$ Doctor
5. α_7 Doctor $<: \alpha_6$ Doctor
6. $\alpha_7 \neq$ lost
7. $\alpha_7 \neq$ any
8. $\alpha_6 \neq$ lost
9. α_9 Doctor $<: \alpha_8$ Doctor
10. $\alpha_9 \neq$ lost
11. $\alpha_9 \neq$ any
12. $\alpha_8 \neq$ lost
13. α_6 Doctor $<: \alpha_5$ Doctor

Figure 3.5. Mandatory constraints generated for example program.

gives rise to a set of constraints over the constraint variables within it. These mandatory constraints which correspond one-to-one to the type rules of GUT must be satisfied by a typing in order for it to be deemed legal.

There are five kinds of constraints. In the following expressions u can be either a concrete modifier or a constraint variable, whereas α represents a constraint variable.

Subtype ($u_1 <: u_2$): A subtype constraint ensures that u_1 will be assigned an ownership modifier that is a subtype of the ownership modifier assigned to u_2. Subtype constraints are used for assignments and for pseudo-assignments (parameter passing, result passing, type variable bound checks).

Adaptation ($u_1 \triangleright u_2 = \alpha_3$): An adaptation constraint enforces that the viewpoint adaptation of variable u_2 from the viewpoint expressed by u_1 results in α_3.

Equality ($u_1 = u_2$): An equality constraint dictates that two modifiers are the same.

They are used to handle method overriding and type argument subtyping, which are both invariant.

Inequality ($u_1 \neq u_2$): An inequality constraint ensures that two modifiers are not equal. For example, the type system forbids the `lost` modifier on the left-hand side of an assignment. The type system also forbids the `any` modifier for the receiver of field updates, if the owner-as-modifier discipline is enforced.

Comparable ($u_1 <:> u_2$): A comparable constraint expresses that two ownership modifiers are not incompatible, that is, one could be a subtype of the other. Comparable constraints are used for casts.

Fig. 3.5 shows the mandatory constraints which are generated for the example input program in Fig. 3.4. A detailed explanation of how GUT type constraints are generated from different program constructs is deferred to the next section.

Breakable Constraints For a given set of type constraints, there may be multiple satisfying assignments. For a completely unannotated program, one of these assignments includes the trivial one that assigns `peer` to all variables. As mentioned previously, this gives rise to a flat ownership structure which is typically not desired. There are many factors which influence the ownership structure desired by the programmer. A deeper structure provides better encapsulation so it is generally preferable but it restricts sharing. On the other hand, the types used in method signatures influence which clients may call the method, so it is typically preferable for method parameters to have the less restrictive `any` modifier. To reflect these and other kinds of design considerations, preferences for a constraint variable's assignment can be expressed by adding a breakable, weighted equality constraint for it. An example of a predefined heuristic which prefers solutions which deep ownership structures and generally applicable methods is as follows:

- For field types, the weight for $\alpha =$ `rep` is 80.
- For parameter types, the weight for $\alpha =$ `any` is 150.
- For return types, the weight for $\alpha =$ `rep` is 30.
- For class and method type variable bounds, the weight for $\alpha =$ `any` is 200.

Such a heuristic may be adapted by a user either globally or for individual variables. The programmer is free to have as many distinct weights as is required to express his preferences. There is also no upperbound on the value that a weight can take.

Constraint	Encoding
$\alpha_1 <: \alpha_2$	$(\beta_1^{any} \Rightarrow \beta_2^{any}) \ \wedge \ (\beta_2^{peer} \Rightarrow \beta_1^{peer}) \ \wedge$ $(\beta_2^{rep} \Rightarrow \beta_1^{rep}) \ \wedge \ (\beta_1^{lost} \Rightarrow (\beta_2^{lost} \vee \beta_2^{any}))$
$\alpha_1 \rhd \alpha_2 = \alpha_3$	$(\beta_1^{peer} \wedge \beta_2^{peer} \Rightarrow \beta_3^{peer}) \ \wedge \ (\beta_1^{rep} \wedge \beta_2^{peer} \Rightarrow \beta_3^{rep}) \ \wedge$ $(\beta_2^{any} \Rightarrow \beta_3^{any}) \ \wedge \ (\beta_2^{lost} \Rightarrow \beta_3^{lost}) \ \wedge \ (\beta_1^{any} \wedge \neg\beta_2^{any} \Rightarrow \beta_3^{lost}) \ \wedge$ $(\beta_1^{lost} \wedge \neg\beta_2^{any} \Rightarrow \beta_3^{lost}) \ \wedge \ (\beta_2^{rep} \Rightarrow \beta_3^{lost})$
$\alpha_1 = \alpha_2$	$(\beta_1^{peer} \Rightarrow \beta_2^{peer}) \ \wedge \ (\beta_1^{rep} \Rightarrow \beta_2^{rep}) \ \wedge$ $(\beta_1^{lost} \Rightarrow \beta_2^{lost}) \ \wedge \ (\beta_1^{any} \Rightarrow \beta_2^{any})$
$\alpha_1 \neq \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg\beta_2^{peer}) \ \wedge \ (\beta_1^{rep} \Rightarrow \neg\beta_2^{rep}) \ \wedge$ $(\beta_1^{lost} \Rightarrow \neg\beta_2^{lost}) \ \wedge \ (\beta_1^{any} \Rightarrow \neg\beta_2^{any})$
$\alpha_1 <:> \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg\beta_2^{rep}) \ \wedge \ (\beta_1^{rep} \Rightarrow \neg\beta_2^{peer})$

Figure 3.6. The encoding for each of the five kinds of constraints, a figure taken from [1].

Reduction to Partial Weighted Max-SAT In the final step, the constraint system is reduced to a partial weighted Max-SAT instance and a partial weighted Max-SAT solver is used to find a solution. The reduction is as follows:

1. For each constraint variable α_i in the inference domain, create four boolean variables, β_i^{peer}, β_i^{rep}, β_i^{any}, and β_i^{lost} and generate the following hard clauses to ensure that exactly one of these four boolean variables is assigned true.

$$(\beta^{peer} \vee \beta^{rep} \vee \beta^{any} \vee \beta^{lost}) \wedge \neg(\beta^{peer} \wedge \beta^{rep}) \wedge \neg(\beta^{peer} \wedge \beta^{any}) \wedge$$
$$\neg(\beta^{peer} \wedge \beta^{lost}) \wedge \neg(\beta^{rep} \wedge \beta^{any}) \wedge \neg(\beta^{rep} \wedge \beta^{lost}) \wedge \neg(\beta^{lost} \wedge \beta^{any})$$

   Note that for every variable that will be explicitly inserted into the program, `lost` is forbidden and the encoding of the variable is accordingly simplified.

2. For each mandatory constraint, generate hard clauses according to Fig. 3.6.

3. For each weighted, breakable constraint, generate single literal soft clauses having the same weight. For example, $\alpha_i =$ `peer` with a weight of 80, translates to β_i^{peer} with a weight of 80.

Since each of the steps in the above reduction can be performed in linear time, it is a polynomial time reduction. To show that the above reduction from tunable type inference (TTI) to partial weighted Max-SAT is sound, we have to show that an instance

of the former has a solution if and only if the corresponding instance of the latter has a solution.

Assume X is a satisfiable TTI instance and we have the optimal assignment. We can construct the optimal assignment for the corresponding SAT instance as follows: for each i, set the β_i's according to the modifier assigned to α_i, for example, if $\alpha_i = \texttt{rep}$, then set $\beta_i^{rep} = 1$, $\beta_i^{peer} = 0$, $\beta_i^{any} = 0$ and $\beta_i^{lost} = 0$. Since the TTI assignment satisfied all the mandatory type constraints, the corresponding SAT assignment will also satisfy all the mandatory SAT clauses. Furthermore, since there is a direct mapping from each breakable TTI constraint to the corresponding breakable SAT constraint, it is also observed that if the TTI assignment yields the optimum weight, then so does the corresponding SAT assignment.

Assume Y is a satisfiable instance of partial weighted Max-SAT and we have the optimal assignment. We can construct the optimal assignment for the corresponding TTI instance as follows: for each i, set α_i according to the assignment of the β_i's, for example, if $\beta_i^{rep} = 1$ set $\alpha_i = \texttt{rep}$. Since the SAT assignment satisfied all the mandatory SAT clauses, the corresponding TTI assignment will satisfy all the mandatory type constraints. Furthermore, since there is a direct mapping from each breakable SAT constraint to the corresponding breakable TTI constraint, it is also observed that if the SAT assignment yields the optimum weight, then so does the corresponding TTI assignment.

Therefore, the above reduction from tunable type inference to partial weighted Max-SAT is correct.

3.3 Constraint Generation

In this section we will outline the rules with which the mandatory GUT type constraints are generated from source code.

Programming Language Fig. 3.7 outlines the syntax of the programming language and the naming conventions [17]. $\overline{A}$ denotes a sequence of A elements. P denotes a program, and is a sequence of class declarations. P is available in all judgements. Cls denotes a class declaration, naming the class and its superclass, along with their type parameters and type arguments, respectively, and consists of field and method declarations. fd denotes a field declaration and is composed of a type and an identifier. md denotes a method declaration and consists of the method purity, method type

$$
\begin{array}{rcl}
P & ::= & \overline{Cls} \\
Cls & ::= & \texttt{class}\ Cid\langle\overline{TP}\rangle\ \texttt{extends}\ C\langle\overline{T}\rangle\ \{\ \overline{fd}\ \overline{md}\ \} \\
TP & ::= & X\ \texttt{extends}\ N \\
md & ::= & p\ \langle\overline{TP}\rangle\ T_r\ m(\overline{T\ pid})\ \{\ e\ \} \\
e & ::= & \texttt{null} \mid x \mid \texttt{new}\ N() \mid e.f \mid e_0.f{:=}e_1 \mid \\
 & & e_0.\langle\overline{T}\rangle m(\overline{e}) \mid (N)\ e \\
u & ::= & \alpha \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{any} \mid \texttt{lost} \mid \texttt{self}
\end{array}
\qquad
\begin{array}{rcl}
C & ::= & Cid \mid \texttt{Object} \\
fd & ::= & T\ f; \\
p & ::= & \texttt{pure} \mid \texttt{impure} \\
T & ::= & N \mid X \\
N & ::= & u\ C\langle\overline{T}\rangle \\
x & ::= & pid \mid \texttt{this}
\end{array}
$$

pid	parameter identifier	f	field identifier
m	method identifier	Cid	class identifier
α	constraint variable identifier	X	type variable identifier

Figure 3.7. Syntax of the programming language, a figure taken from [1].

parameters if any, return type, method name, formal parameter declarations, and a method body. e denotes an expression and can be the `null` literal, a method parameter access, object creation, field read, field update, method call or a cast.

T denotes a type and is either a non-variable type N or a type variable X. A non-variable type N consists of an ownership modifier u and a possibly-parametrized class C. Ownership modifiers u include the concrete modifiers `peer`, `rep`, `any`, `lost`, and `self`, as well as the constraint variables, α. α, `lost` or `self` are not part of the surface syntax. The omission of modifiers is permitted; constraint variables are generated for all omitted modifiers.

Constraint Generation The rules of constraint generation are summarized in Fig. 3.8. These rules make use of helper judgements and functions that lift constraints from single modifiers to types; these are defined in Fig. 3.9. We first discuss the main rules and then move on to the helper functions. Since GUT type inference is applicable only for compilable Java programs, the rules in Fig. 3.8 do not encode all Java type rules but rather only give constraints for the additional GUT checks needed.

Γ denotes an environment which maps type variables of the enclosing class and method to their upper bounds and variables to their types. The notations $\Gamma(X)$ and $\Gamma(x)$ are used to look-up the upper bound of a type variable and the type of a variable, respectively. The helper function `env` defines the environment necessary for checking class and method declarations.

The constraints for a class, field, method parameter and method declaration, arise from

Environment: $\Gamma = \{\overline{X \mapsto N}; \overline{x \mapsto T}\}$

Class declaration: $\boxed{\vdash Cls : \Sigma}$

$$\frac{\begin{array}{c}\text{env}(Cid, \overline{TP}) = \Gamma \quad \Gamma \vdash \overline{fd} : \Sigma_f \quad \Gamma \vdash \overline{md} : \Sigma_m \\ \Gamma \vdash \texttt{self}\ C\langle\overline{T}\rangle, \text{bounds}(\overline{TP})\ \text{OK} : \Sigma_t \\ \Sigma = \Sigma_f \cup \Sigma_m \cup \Sigma_t\end{array}}{\vdash \texttt{class}\ Cid\langle\overline{TP}\rangle\ \texttt{extends}\ C\langle\overline{T}\rangle\ \{\ \overline{fd}\ \overline{md}\ \} : \Sigma}$$

Field and method parameter declaration: $\boxed{\Gamma \vdash T\ f : \Sigma}$, $\boxed{\Gamma \vdash T\ pid : \Sigma}$

$$\frac{\Gamma \vdash T\ \text{OK} : \Sigma}{\Gamma \vdash T\ f : \Sigma} \qquad \frac{\Gamma \vdash T\ \text{OK} : \Sigma}{\Gamma \vdash T\ pid : \Sigma}$$

Method declaration: $\boxed{\Gamma \vdash md : \Sigma}$

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

Expressions: $\boxed{\Gamma \vdash e : T, \Sigma}$

$$\frac{\Gamma \vdash e : T_0 : \Sigma_0 \quad \Gamma \vdash T_0 <: T : \Sigma_1}{\Gamma \vdash e : T, \Sigma_0 \cup \Sigma_1} \qquad \frac{}{\Gamma \vdash \texttt{null} : T, \emptyset} \qquad \frac{}{\Gamma \vdash x : \Gamma(x), \emptyset} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0, \Sigma_0 \\ \Gamma \vdash N\ \text{OK} : \Sigma_t \\ \Gamma \vdash N <:> T_0 : \Sigma_c \\ \Sigma = \Sigma_0 \cup \Sigma_t \cup \Sigma_c\end{array}}{\Gamma \vdash (N)\ e_0 : N, \Sigma}$$

$$\frac{\begin{array}{c}\Gamma \vdash N\ \text{OK} : \Sigma_0 \\ \Sigma_1 = \{\text{om}(N) \neq \{\texttt{lost}, \texttt{any}\}\}\end{array}}{\Gamma \vdash \texttt{new}\ N() : N, \Sigma_0 \cup \Sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : N_0, \Sigma_0 \qquad \Gamma \vdash e_1 : T_1, \Sigma_1 \\ \text{fType}(N_0, f) = T_2, \Sigma_2 \qquad \Gamma \vdash T_1 <: T_2 : \Sigma_3 \\ \Sigma_4 = \{\texttt{lost} \notin T_2\} \\ \Sigma_5 = \{\boxed{\text{om}(N_0) \neq \{\texttt{lost}, \texttt{any}\}}\}\end{array}}{\Gamma \vdash e_0.f{:=}e_1 : T_2, \bigcup_{i=0}^{i=5} \Sigma_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : N_0, \Sigma_0 \\ \text{fType}(N_0, f) = T, \Sigma_1\end{array}}{\Gamma \vdash e.f : T, \Sigma_0 \cup \Sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : N_0, \Sigma_0 \qquad \Gamma \vdash \overline{e_a} : \overline{T_a}, \Sigma_1 \\ \text{mType}(N_0, m, \overline{T}) = p\ \langle\overline{TP}\rangle\ T_r\ m(\overline{T_p\ pid}), \Sigma_2 \\ \Gamma \vdash \overline{T_a} <: \overline{T_p} : \Sigma_3 \\ \Sigma_4 = \{\texttt{lost} \notin (\overline{T_p}, \text{bounds}(\overline{TP}))\} \\ \Gamma \vdash \overline{T}\ \text{OK} : \Sigma_5 \qquad \Gamma \vdash \overline{T} <: \text{bounds}(\overline{TP}) : \Sigma_6 \\ p = \texttt{impure} \Rightarrow \Sigma_7 = \{\boxed{\text{om}(N_0) \neq \{\texttt{lost}, \texttt{any}\}}\} \\ p = \texttt{pure} \Rightarrow \Sigma_7 = \emptyset\end{array}}{\Gamma \vdash e_0.\langle\overline{T}\rangle m(\overline{e_a}) : T_r, \bigcup_{i=0}^{i=7} \Sigma_i}$$

Figure 3.8. Constraint generation rules, a figure taken from [1].

the constraints of their components. The well-formed type (OK) helper judgement helps to ensure the well-formedness of types. Note that the environment for a method declaration is extended with the method type variables and the method parameters. In the case of function overriding, if the current method is overriding a method in a superclass, it is necessary for the parameter and return types to be consistent. This requirement is reflected by the constraint set, Σ_2, which consists of equality constraints between the types in the current method signature and a directly overridden method signature.

There are eight judgements for expressions. For an object creation expression, the main ownership modifier cannot be `lost` or `any`, rather `peer` or `rep` must be inferred in order to give the new object a specific location in the ownership topology. Helper functions `fType` and `mType` (to be discussed below) yield the field type, and method signature respectively, after viewpoint adaptation, and additional constraints that encode the necessary adaptations of modifiers. As mentioned earlier, in order to ensure soundness `lost` has to be forbidden from the left-hand side of assignments. For this reason, the adapted field type, the adapted method parameter types, and the method type variable bounds are forbidden from being `lost`. These constraints ensure that modifications are only allowed if the ownership is statically known. When the option of owner-as-modifier is enforced, the rules for field updates and for impure methods give rise to additional constraints: the main modifier of the receiver expression cannot be `lost` or `any`.

The $\Gamma \vdash N <:> T_0 : \Sigma_c$ clause of the cast rule requires explanation. A cast is a type loophole that indicates that the program's behaviour is beyond the reasoning capabilities of the type system. A program which contains a cast may fail the runtime check. Generic Universe Types also supports casts: downcasts which specialize ownership information (these are casts from `any` to `peer` or `rep`) and require runtime checks. The GUT inference is never permitted to insert a new cast into the program as this would defeat the purpose of static ownership type checking. However, for existing casts, it is permitted to choose arbitrary ownership modifiers and therefore an existing cast might fail at runtime either because of the base language check or because of the ownership check. For example, if the inferred modifiers of variables `x` and `o` are `peer` and `any` respectively, then the constraint for the expression `x = (Person) o` infers `peer` as the modifier for the cast in order to make the assignment type-correct. An alternative would be for the static inference to choose modifiers in such a way so as to guarantee that the runtime ownership check at each cast is passed. This can be accomplished by changing "<:>" to "=". Subsumption of the expression type could then still be used to cast to a supertype, which is guaranteed to succeed.

Notation:

$\mathrm{om}(u\ C\langle\overline{T}\rangle) \equiv u$ $\qquad$ $(u \neq \{u_1, u_2, \ldots\}) \equiv (u \neq u_1, u \neq u_2, \ldots)$

$\mathrm{bounds}(\overline{X\ \texttt{extends}\ N}) \equiv \overline{N}$ $\qquad$ $(u \notin u'\ C\langle\overline{T}\rangle) \equiv (u \neq u' \wedge u \notin \overline{T})$

Environment definitions:

$\mathrm{env}(Cid, \overline{X\ \texttt{extends}\ N}) = \{\overline{X \mapsto N}; \texttt{this} \mapsto \texttt{self}\ Cid\langle\overline{X}\rangle\}$

$\mathrm{env}(\{\overline{X_c \mapsto N_c}; \overline{x \mapsto T}\}, \overline{X\ \texttt{extends}\ N}, \overline{T_p\ pid}) = \{\overline{X_c \mapsto N_c}, \overline{X \mapsto N}; \overline{x \mapsto T}, \overline{pid \mapsto T_p}\}$

Ownership modifier - type adaptation: $\boxed{u \triangleright T = T' : \Sigma}$

$$\frac{}{u \triangleright X = X : \emptyset} \qquad \frac{\Sigma_0 = \{u \triangleright u' = \alpha\} \quad \mathrm{fresh}(\alpha) \quad u \triangleright \overline{T} = \overline{T'} : \Sigma_1}{u \triangleright u'\ C\langle\overline{T}\rangle = \alpha\ C\langle\overline{T'}\rangle : \Sigma_0 \cup \Sigma_1}$$

Type - type adaptation: $\boxed{\Gamma \vdash N \triangleright T = T' : \Sigma}$

$$\frac{u \triangleright T = T_1 : \Sigma \quad T_1\,[\,\overline{T}/\overline{X}\,] = T' \quad \mathrm{typeVars}(C) = \overline{X}}{u\ C\langle\overline{T}\rangle \triangleright T = T' : \Sigma}$$

Subtyping: $\boxed{\Gamma \vdash T <: T' : \Sigma}$

$$\frac{\Sigma = \{u <: u', \overline{T} = \overline{T'}\}}{\Gamma \vdash u\ C\langle\overline{T}\rangle <: u'\ C\langle\overline{T'}\rangle : \Sigma} \qquad \frac{C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{T_1}\rangle \quad u\ C\langle\overline{T}\rangle \triangleright \overline{T_1} = \overline{T'}, \Sigma}{\Gamma \vdash u\ C\langle\overline{T}\rangle <: u\ C'\langle\overline{T'}\rangle : \Sigma}$$

$$\frac{}{\Gamma \vdash X <: X : \emptyset} \qquad \frac{}{\Gamma \vdash X <: \Gamma(X) : \emptyset} \qquad \frac{\Gamma \vdash T <: T_1 : \Sigma_1 \quad \Gamma \vdash T_1 <: T' : \Sigma_2}{\Gamma \vdash T <: T' : \Sigma_1 \cup \Sigma_2}$$

Well-formed type: $\boxed{\Gamma \vdash T\ \mathrm{OK} : \Sigma}$

$$\frac{\Gamma \vdash \overline{T}\ \mathrm{OK} : \Sigma_0 \quad \mathrm{typeBounds}(u\ C\langle\overline{T}\rangle) = \overline{T'}, \Sigma_1 \quad \Gamma \vdash \overline{T} <: \overline{T'} : \Sigma_2}{\Gamma \vdash u\ C\langle\overline{T}\rangle\ \mathrm{OK} : \bigcup_{i=0}^{i=2} \Sigma_i} \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ \mathrm{OK} : \emptyset}$$

Adaptation of a field type: $\boxed{\mathrm{fType}(N, f) = T, \Sigma}$

$\mathrm{fType}(u\ C\langle\overline{T}\rangle, f) = T', \Sigma$ $\quad$ where $\quad \mathrm{fType}(C, f) = T$, $u\ C\langle\overline{T}\rangle \triangleright T = T' : \Sigma$

Adaptation of a method signature: $\boxed{\mathrm{mType}(N, m, \overline{T}) = p\ \overline{TP}\ T_r\ m(\overline{T_p\ pid}), \Sigma}$

$\mathrm{mType}(N, m, \overline{T'}) = p\ \langle\overline{X\ \texttt{extends}\ N'}\rangle\ T'_r\ m(\overline{T'_p\ pid}), \Sigma_b \cup \Sigma_r \cup \Sigma_p$

where $N = u\ C\langle\overline{T}\rangle$

$\mathrm{mType}(C, m) = p\ \langle\overline{X\ \texttt{extends}\ N_b}\rangle\ T_r\ m(T_p\ pid)$

$N \triangleright \overline{N_b} = \overline{N_0} : \Sigma_b \qquad N \triangleright T_r = T_{r0} : \Sigma_r \qquad N \triangleright \overline{T_p} = \overline{T_{p0}} : \Sigma_p$

$\overline{N_0}[\overline{T'}/\overline{X}] = \overline{N'} \qquad T_{r0}[\overline{T'}/\overline{X}] = T'_r \qquad \overline{T_{p0}}[\overline{T'}/\overline{X}] = \overline{T'_p}$

Adaptation of type bounds: $\boxed{\mathrm{typeBounds}(N) = \overline{N'}, \Sigma}$

$\mathrm{typeBounds}(u\ C\langle\overline{T}\rangle) = \overline{N'}, \Sigma$ $\quad$ where $\quad \texttt{class}\ C\langle\overline{X\ \texttt{extends}\ N}\rangle \ldots \in P$, $u\ C\langle\overline{T}\rangle \triangleright \overline{N} = \overline{N'}, \Sigma$

Look-up of class type variables: $\boxed{\mathrm{typeVars}(N) = \overline{X}}$

$\mathrm{typeVars}(u\ C\langle\overline{T}\rangle) = \overline{X}$ $\quad$ where $\quad \texttt{class}\ C\langle\overline{X\ \texttt{extends}\ N}\rangle \ldots \in P$

Figure 3.9. Helper functions and judgements, a figure taken from [1].

Helper Functions We now discuss the helper judgements and function in Fig. 3.9 which support the main judgements in Fig. 3.8. The rules show how constraints are lifted from single modifiers to types.

Function `om` returns the ownership modifier of a non-variable type. Function `bounds` returns the upper bound types of type parameter declarations. A compact notation is used to compare one ownership modifier against a set of modifiers and to ensure that a modifier does not appear in a type. Function `env` defines the environment based on the program's class and method declarations.

Each judgement can also be used on a sequences of elements by applying the judgement to each individual element and then combining results. For simplicity this is not shown in Fig. 3.9.

Viewpoint adaptation can be lifted from single modifiers to types using two judgements. One adapts a type from an ownership modifier and the other adapts a type from the viewpoint of a non-variable type. A type is adapted from the viewpoint of an owernship modifier to `this` giving an adapted type and a constraint set. There are two cases: (1) No constraint is generated to adapt type variables X as they do not need to be adapted, (2) The constraints to adapt a non-variable type $u'\ C\langle\overline{T}\rangle$ from viewpoint u consist of the constraint for combining u with the main modifier u'. The result is captured in a fresh constraint variable α. The type arguments are then recursively adapted. A type is adapted from the viewpoint of a non-variable type to `this`, by first adapting the type using the main modifier u and then substituting the type arguments $\overline{T}$ for the type variables $\overline{X}$. Function typeVars gives the type variables defined by a class. The notation $T[\,\overline{T}/\overline{X}\,]$ is used to substitute type arguments $\overline{T}$ for occurrences of type variables $\overline{X}$ in T.

The subtyping judgement determines the set of constraints which must hold in order for two types to be subtypes. Of the five rules in this judgement, the second one is the most interesting. It derives a subtyping relationship from a subclassing relationship by adapting the type arguments from the superclass to the particular subtype instantiation. The subclassing relationship $\sqsubseteq$ is the reflexive and transitive closure of the `extends` relationship of the classes; it is defined over instantiated classes $C\langle\overline{T}\rangle$, as defined in GUT [3].

Fig. 3.9 does not show lifted versions of equality and comparable constraints. The former is lifted to types by simple recursion. The latter is applied to two non-variable types by first going to a common superclass and then generating a comparable constraint for the two main modifiers and equality constraints for the type arguments.

The well-formed type (OK) judgement decides when a type T is well-formed in an

environment Γ giving the constraints Σ. The well-formedness of environments is not shown in Fig. 3.9; they are just the well-formedness for all involved types.

Finally, we discuss the overloaded helper functions `fType`, `mType`, and `typeBounds` which are defined as follows. Function `fType(C, f)` yields the declared field type of field f in class C or a superclass of C. It yields only a type, but no constraints.

The overloaded function `fType(N, f)` (taking a non-variable type rather than a class as first argument) determines the type of field f adapted from viewpoint N to `this`. It results in an adapted field type and constraints on the constraint variables of the viewpoint and the constraint variables for the declared type.

Function `mType(C, m)` yields the declared method signature of method m in class C or a superclass of C. The overloaded function `mType (N, m,` $\overline{T}$`)` determines the method signature of method m adapted from viewpoint N to `this` and substituting method type arguments for their type variables. It results in an adapted method signature and constraints on the constraint variables of the viewpoint and the constraint variables for the declared parameter, return, and type variable bound types, respectively.

Function `typeBounds(u C`$\langle\overline{T}\rangle$`)` yields the upper bounds of the type variables of class C adapted from the non-variable type $u\ C\langle\overline{T}\rangle$ to `this` and a set of constraints.

3.4 Implementation and Evaluation

In this section we summarize the implementation and experience of [1]. They implemented static inference on top of the Checker Framework [14], a pluggable type checking framework built on top of the JSR 308 branch of the OpenJDK compiler. The implementation consists of approximately 4400 non-comment, non-blank lines of Scala code. The tool is modular and only generates constraints for the part of the program that is supplied as input. For the remainder of the program, in particular for the JDK libraries, the tool uses the default modifier `peer`.

They tested the inference on four real-word, open source tools developed by external developers. The four benchmarks are: (1) OpenJDK's implementation of the zip and gzip compression algorithms, taken from OpenJDK 7 build 138, (2) javad, a Java class file disassembler, (3) JDepend, a quality metrics tool, and (4) Classycle, a Java class dependency analyzer. For each benchmark, two solutions were inferred, with and without the owner-as-modifier option.

Benchmark	SLOC	Constraint Size			CNF Size			Timing (seconds)			
		vars	constraints		vars	clauses		topol.		encap.	
			topol.	encap.		topol.	encap.	gen	solve	gen	solve
1. zip	2611	455	2411	2949	4656	13639	14063	4.5	1.1	4.5	1.1
2. javad	1846	364	2571	3113	4988	14989	15333	3.5	1.0	3.6	1.0
3. jdepend	2460	824	4868	6024	9752	28110	29176	5.1	1.4	5.8	1.5
4. classycle	4658	1548	8726	10242	17756	53062	54380	6.0	1.8	6.2	2.0

Figure 3.10. Size and timing results, a figure taken from [1].

Fig. 3.10 shows a summary of the size and timing information of these experiments. SLOC denotes the number of non-blank, non-comment lines. The constraint size columns give the number of constraint variables and constraints in the program. The CNF size gives the number of boolean variables and clauses in the SAT encoding. Finally, the timing columns give the time for generating (gen) and solving (solve) the constraints. Each of the runs were executed three times and the median was reported. The number of constraints and clauses and the timing is further sub-divided into whether annotations for only the topology or also for enforcing the encapsulation discipline should be inferred. This choice does not affect the number of constraint variables or boolean variables.

To evaluate the correctness of the tool, the inferred annotations were inserted into the source code and the GUT type checker was run on the program. For all cases, the type checker verified correctness.

To evaluate the usefulness of inferred annotations, the inference results were manually examined. This examination seemed to indicate that the inference accurately reflected the ownership properties of the original programs [1].

To evaluate the scalability of the tool, the tool was applied to JabRef, a bibliography management tool consisting of around 74000 SLOC. The inference generated 24402 variables and 248858 constraints, which were then encoded into 521152 boolean variables and 1606319 CNF clauses. Generation of the constraint system took a total of 41 seconds and solving the system took a total of 66 seconds, of which 42 seconds were spend in the SAT solver. The software used to insert the annotations back in the code was unable to handle such a large amount of annotations and crashed, so the GUT checker could not be used to verify the results.

Our Comments Based on our empirical experience with SAT solvers, the timing results in Figure 2.6 suggests to us that the tool ran efficiently on these four benchmarks.

In other words, these particular benchmarks seem to be easy instances for the inference. Some natural questions which then arise and which the next chapter attempts to answer are: Are all instances of the inference easy? If yes, is there a more efficient way of solving the inference? If not, what is an example of a hard instance?

Chapter 4

Complexity Analysis

This chapter presents our contributions. We first show that the decision version of the tunable type inference (TTI) problem for GUT is **NP**-complete. We then prove that the problem is also difficult to approximate to within a certain factor. Three restricted but practical versions of the problem are examined and their complexity discussed. We report our efforts to explore the possible sources of complexity and to gain further insight into the characteristics of a hard instance of the TTI problem.

4.1 Type Inference for GUT is NP-Complete

The tunable type inference problem for GUT is inherently difficult. The approach we take to prove this is as follows: We first define a new problem, Monotone-2-SAT-Max, and prove that it is **NP**-complete. We then reduce Monotone-2-SAT-Max to TTI, thereby proving that the latter is **NP**-complete too. Before proceeding, it is worth mentioning the following properties of the TTI problem which can be easily overlooked but have to be considered when devising the proof.

- While every compilable program P gives rise to a set of mandatory type constraints, not every set of mandatory constraints can be mapped to a valid program. For example, from the constraint generation rules outlined in Sec. 3.3, it can be seen that any rule which gives rise to the constraint $\alpha \neq$ `any` also gives rise to the constraint $\alpha \neq$ `lost`. Hence we can conclude that any set of mandatory constraints which includes the former but not the latter cannot be mapped to any valid program. For this reason it is appropriate to define an instance of the

TTI problem to be $< P, B, k >$ where P is a valid program from which the set of mandatory constraints can be extracted, B is the set of breakable, weighted constraints provided by the user and k is a positive integer representing the desired lower-bound on the weight achieved.

- The set of breakable, weighted constraints B is limited to equality constraints in which constraint variables are equal to fixed modifiers. For example, $\alpha =$ `any` is a valid breakable constraint but $\alpha_i \neq \alpha_j$ is not.

- The set B is provided by the user and reflects the user's programming intentions. Since the modifier `lost` is not a part of the surface syntax, we do not expect the user to prefer it for any constraint variables. In other words, none of the breakable constraints will involve a `lost` modifier.

- The constraint variables which result from viewpoint adaptations are implicit and do not represent any actual annotations to the source code so we do not expect the user to specify preferences for them. In other words, the constraint variables arising from viewpoint adaptations will not be involved in any breakable constraints.

- From the constraint generation rules, it can be seen that any inequality mandatory constraint must involve a constraint variable and a fixed modifier. For example, $\alpha \neq$ `lost` is a possible inequality constraint but $\alpha_i \neq \alpha_j$ is not.

Monotone-2-SAT-Max We introduce a new problem called Monotone-2-SAT-Max and define it as follows.

Definition 4.1. *Monotone-2-SAT-Max* $= \{< \sigma, m, t >: \exists$ *an assignment that satisfies* σ *and achieves a weight of at least* t *under the mapping* $m\}$ *where*

- σ *is a boolean formula in 2-CNF-SAT format with these additional conditions: (1) both literals in each disjunction must be positive (hence the word monotone), (2) they must be distinct. Examples of disjunctions which are not permitted are* $x \vee -y$ *and* $x \vee x$.

m *is a mapping of the assignment of variables in* σ *to a positive integer weight e.g.* $x = 0$ *has a weight of 10 and* $x = 1$ *has a weight of 5. If an assignment has not been given an explicit weight, the default weight of 0 is assumed. This mapping is additive, so for example if* $x_1 = 1$ *has a weight of 10 and* $x_2 = 1$ *has a weight of 20, then the total weight achieved by setting both variables to true is 30.*

t *is a non-negative integer denoting the desired lower bound on the weight to be achieved*

Theorem 4.1. *Monotone-2-SAT-Max is* ***NP****-complete.*

Proof. Monotone-2-SAT-Max is in **NP**; a suitable certificate is a satisfying assignment to the boolean variables in σ. The size of the certificate is equal to the number of boolean variables and it can be used to verify a 'yes' instance in linear time. Monotone-2-SAT-Max is **NP**-hard. This can be proved by a reduction from a well-known **NP**-complete problem called CLIQUE. A clique is a subset of vertices of an undirected graph such that there exits an edge between every pair of vertices in the subset. The decision version of the CLIQUE problem is defined as follows:

Definition 4.2. *CLIQUE* $= \{\langle G = \langle V, E\rangle, d\rangle : G$ *is a graph with vertices* V *and edges* E, $d \geq 2$ *is an integer, and* $\exists$ *a subset* $V' \subseteq V$ *such that* $|V'| \geq d$ *and for every pair of vertices* $(u, v) \in V'$, *there exists an edge* $\langle u, v\rangle \in E\}$

The following steps describe the operation of the function f which maps a CLIQUE instance to a Monotone-2-SAT-Max instance.

1. Create a dummy boolean variable z.
2. For every vertex $v \in V$ create a boolean variable x_v and form the clause $x_v \vee z$.
3. For every edge $(u, v) \notin E$ form the clause $x_u \vee x_v$.
4. σ is the conjunction of all the clauses formed in steps 2 and 3.
5. The mapping m is: $x_i = 0$ has a weight of 1, for $1 \leq i \leq |V|$. All remaining assignments have a weight of 0.
6. $t = d$

To show that the above conversion can be done in polynomial time, we have to show this for step 3. It is obvious that the other steps can be performed in either constant or linear time. Consider step 3. In a graph $G = (V, E)$, the maximum possible number of edges there can be is $\binom{V}{2}$, which is $O(|V|^2)$. To scan the list E to check whether an edge exists within it or not takes at most $O(|E|)$ time. So step 3 can be performed in time $O(|V|^2|E|)$ which is polynomial in the size of the input. Hence the function f is polynomial time computable. $x_i = 1$ in the SAT domain implies that the vertex i is not in the clique and $x_i = 0$ in the SAT domain implies that the vertex i is in the clique. To complete the proof of the reduction, we must now prove that $\langle G, d\rangle \in$ CLIQUE $\iff f(\langle G, d\rangle) \in$ Monotone-2-SAT-Max or equivalently, the following two claims:

Claim 4.1. *If the CLIQUE instance is a 'yes' instance then the corresponding Monotone-2-SAT-Max instance is a 'yes' instance.*

Proof. Assume that the CLIQUE instance is a 'yes' instance i.e. the graph $G = (V, E)$ has a clique of size at least d, and we know the vertices of one such clique V'. We can construct a valid assignment to σ in the following way: for every vertex $v \in V'$ set $x_v = 0$, else set $x_v = 1$. Set $z = 1$. Since the clique has at least d members, at least d of the x_v's will be set to 0. Hence the weight achieved by this assignment will be at least $t = d$. We must now show that the assignment satisfies the boolean formula. It is obvious that this assignment will satisfy all the clauses formed in step 1 of the reduction. Now consider the clauses formed in step 2. They are of the kind $x_v \vee x_u$ where $(u, v) \notin E$. By the definition of a clique, at least one of the two vertices u and v will not be present in the set V' and hence at least one of x_v and x_u will be set to 1, thereby satisfying all the clauses formed in step 2 of the reduction. Hence this assignment satisfies σ and achieves the desired weight so the Monotone-2-SAT-Max instance is a 'yes' instance. □

Claim 4.2. *If the Monotone-2-SAT-Max instance is a 'yes' instance then the corresponding CLIQUE instance is a 'yes' instance.*

Proof. We prove the above claim by proving the contrapositive. That is, we will show that $\langle G, d\rangle \notin$ CLIQUE $\implies f(\langle G, d\rangle) \notin$ Monotone-2-SAT-Max. Assume that $\langle G, d\rangle \notin CLIQUE$. That is, every possible subset $V' \in V$ such that $|V'| = d$, has at least one pair of vertices (u, v) such that $\langle u, v\rangle \notin E$. In the corresponding Monotone-2-SAT-Max instance, in order to achieve a weight of t, where $t = d$, we require that at least d of the x boolean variables are set to 0. However for every subset of size d of the x boolean variables in σ, there exists at least one pair of variables, (x_u, x_v), in the set,

such that σ contains the disjunction $x_v \vee x_u$. This disjunction will not evaluate to true if both variables are set to 0. Hence the Monotone-2-SAT-Max instance is a 'no' instance.

□

This concludes the proof that the reduction from CLIQUE to Monotone-2-SAT-Max is correct.

□

Reduction of Monotone-2-SAT-Max to TTI Recall the definition of the TTI problem:

TTI = $\{\langle P, B, k\rangle : \exists$ an assignment to the constraint variables of program P that makes P type check and achieves a weight of at least k from the breakable constraints $B\}$

Theorem 4.2. *TTI is **NP**-complete.*

Proof. TTI is in **NP**; a suitable certificate is a satisfying assignment to the constraint variables. The size of this certificate is linear with respect to the number of constraint variables. We can verify that the assignment makes the program type check and that it achieves a weight of at least k from the breakable constraints in linear time. TTI is in **NP**-hard. We prove this by a reduction from Monotone-2-SAT-Max. The following steps describe the operation of the function h which maps a Monotone-2-SAT-Max instance to a TTI instance.

1. Index the boolean variables in σ with the variable i, where $1 \leq i \leq n$, and n is the number of variables in σ.

2. For every disjunction $x_i \vee x_j$ in σ, ensure that $i > j$. To achieve this, you can reorder the two literals in each disjunction if necessary. Since disjunctions are commutative, the reordering will not have any effective change on σ.

3. Start constructing the program P by creating a class C_1 as follows:

```
class C1
{
  any Object field_a;
  any Object field_b;
}
```

4. For $1 \leq i \leq n$, create the following classes in P

```
class C_{i+1} extends C_i
{
  α_i' C_i field_i;

  void helper_{i+1} (any Object parameter_a, any Object parameter_b)
  {
    α_i C_i ob_i = new α_i^temp C_i();
    ob_i.field_b = parameter_b;
    field_i = ob_i;
    field_i.field_a = parameter_a;
  }

}
```

5. For every disjunction $x_i \vee x_j$ in σ, where $i > j$, add the following expression in the helper function $helper_{i+1}$ in class C_{i+1}.

   `any C_i ob_temp_j = ( rep C_i) ob_i.field_j;`

   The above snippet of code is the reason why it is necessary to set up a linked chain of classes. For the object ob_i to have access to the field $field_j$ where j can be any integer such that $i > j$ the class C_i needs to inherit all the classes C_m where $1 \leq m < i$.

6. Use the mapping m of the SAT instance to add the following breakable constraints to B

   $x_i = 1$, weight $w_i \implies \alpha_i =$ `rep`, weight w_i

   $x_i = 0$, weight $w_i' \implies \alpha_i =$ `peer`, weight w_i'

7. $k = t$

8. The desired instance of TTI is $\langle P, B, k \rangle$

Assuming the owner-as-modifier discipline is enforced, the generated program P gives rise to the following mandatory constraints for each i:

- α_i `C_i ob_i = new` α_i^{temp} `C_i();`

  $\alpha_i^{temp} \neq$ `lost`

  $\alpha_i^{temp} \neq$ `any`

$\alpha_i^{temp} <: \alpha_i$

$\alpha_i \neq$ `lost`

- `ob`$_i$`.field_b = parameter_b;`
  `field`$_i$ `= ob`$_i$`;`
  `field`$_i$`.field_a = parameter_a;`

  $\alpha_i <: \alpha_i^{'}$

  $\alpha_i \neq$ `any`

  $\alpha_i \neq$ `lost`

  $\alpha_i^{'} \neq$ `any`

  $\alpha_i^{'} \neq$ `lost`

- `any C`$_i$ `ob_temp`$_j$ `= ( rep C`$_i$`) ob`$_i$`.field`$_j$`;`

  $\alpha_i \triangleright \alpha_j^{'} = z_{ij}$

  $z_{ij} <:>$ `rep`

  z_{ij} is a constraint variable representing the result of the viewpoint adaptation. The comparable constraint is required to prevent z_{ij} from being assigned the modifier `peer`.

Each step of the above reduction can be performed in linear time with respect to the size of the input instance of Monotone-2-SAT-Max so the reduction can be done in polynomial time. In order to prove that the reduction is correct, we must now prove that $y \in$ Monotone-2-SAT-Max $\iff h(y) \in$ TTI or equivalently, the following two claims:

Claim 4.3. *If the TTI instance is a 'yes' instance then the corresponding Monotone-2-SAT-Max instance is a 'yes' instance.*

Proof. Assume the TTI instance is a 'yes' instance and we have the satisfying assignment. We can construct an assignment for the corresponding Monotone-2-SAT-Max instance as follows: For each i, where $1 \leq i \leq n$, if $\alpha_i =$ `rep` in the TTI domain, then set $x_i = 1$ in the SAT domain and if $\alpha_i =$ `peer` in the TTI domain then set $x_i = 0$ in the SAT domain. From the mandatory constraints $\alpha_i \neq any$ and $\alpha_i \neq lost$, we know that every α_i can only take on values of either `rep` or `peer` and hence every boolean variable will be assigned either 1 or 0. Since the TTI assignment achieved a weight of at least k and the set of breakable constraints B corresponds directly to the mapping m, it follows

that the SAT assignment will also achieve a weight of $k = t$. It now remains to show that the assignment is a satisfying one. Consider the following mandatory constraints in the TTI domain:

$\alpha_i^{'} \neq any$

$\alpha_i^{'} \neq lost$

$\alpha_i <: \alpha_i^{'}$

$z_{ij} <:> \texttt{rep}$

$\alpha_i \triangleright \alpha_j^{'} = z_{ij}$

The first three constraints above mean that for every i, $\alpha_i = \alpha_i^{'}$. Since for every j, $\alpha_j^{'}$ can never be assigned the modifier `any`, z_{ij} will never be `any`. Furthermore, z_{ij} cannot be `peer` as this would violate the mandatory constraint $z_{ij} <:> \texttt{rep}$. Therefore z_{ij} can only be assigned `lost` or `rep`. Hence for the viewpoint constraint to hold, the only combinations allowed for $(\alpha_i, \alpha_j^{'})$ or equivalently for (α_i, α_j) are (`rep`, `rep`), (`rep`, `peer`) or (`peer`, `rep`). From this we see that at least one of α_i or α_j will be assigned `rep` which in turn means at least one of x_i and x_j will be assigned 1 in the SAT domain and the disjunction will be satisfied. Since this argument holds for every disjunction, the assignment satisfies σ. Hence the Monotone-2-Sat-Max is a 'yes' instance.

□

Claim 4.4. *If the Monotone-2-SAT-Max instance is a 'yes' instance then the corresponding TTI instance is a 'yes' instance.*

Proof. Let us assume that the Monotone 2-SAT-Max instance is a 'yes' instance and we know the satisfying assignment. We can construct an assignment for the corresponding TTI instance as follows: For every i, if $x_i = 1$ in the SAT domain, set $\alpha_i = \texttt{rep}$ in the TTI domain, and if $x_i = 0$ in the SAT domain, set $\alpha_i = \texttt{peer}$ in the TTI domain. For every i, set $\alpha_i^{temp} = \alpha_i$. For every i, set $\alpha_i^{'} = \alpha_i$. For every z variable set it equal to the result of the viewpoint constraint i.e. set $z_{ij} = \alpha_i \triangleright \alpha_j^{'}$. One can trivially observe that the above assignment will satisfy all the mandatory constraints of the TTI instance.

Furthermore, since the SAT assignment achieved a weight of at least t under the mapping m, and the breakable constraint set B corresponds directly to m, the TTI assignment will also achieve a weight of at least $t = k$. Hence the corresponding TTI instance is a 'yes' instance. □

This concludes the proof that the reduction from Monotone-2-SAT-Max to TTI is correct.

□

We note here that the program P reduced to in the reduction is one which will compile for certain but in some cases may not execute at run time. This however does not affect our proof because we are dealing with static inference; our inference tool should be able to infer annotations for any program which compiles without error. To see why P may not execute, consider the line of code:

```
any C_i ob_temp_j = ( rep C_i) ob_i.field_j;
```

If both α_i and $\alpha_j^{'}$ are inferred to be `rep` then the viewpoint constraint evaluates to `lost`. `lost` is comparable to `rep` and will pass the type check at compile time. However, when trying to cast it to a `rep` object at run-time, an error will occur. If however, α_i is inferred to be `rep` and $\alpha_j^{'}$ to be `peer`, then both the compile time and run time ownership checks will be passed.

4.2 Hardness of Approximation

A standard approach to mitigate the intractability of **NP**-hard problems is to look for efficient algorithms that find approximate solutions, i.e. approximation algorithms. Approximation algorithms are algorithms to find a solution that is guaranteed to be within some factor of the optimum. However, an approximation algorithm cannot be defined for those instances that have no solution. Therefore, we confine ourselves to those instances of TTI that have at least a solution. We establish an inapproximation result for TTI, thereby showing that it is hard to approximate TTI efficiently. We first give the definition of a gap-preserving reduction as this will be used to establish the result.

Definition 4.3. *Let* Π *and* Π' *be two maximization problems and* $\rho, \rho' > 1$. *A gap preserving reduction with parameters* $(c, \rho), (c', \rho')$ *from* Π *to* Π' *is a polynomial time algorithm* f. *For each instance* I *of* Π, f *produces an instance* $I' = f(I)$ *of* Π'. *The optima of* I *and* I', *say* $OPT(I)$ *and* $OPT(I')$ *respectively, satisfy the following properties:*

1. $OPT(I) \geq c \implies OPT(I') \geq c'$
2. $OPT(I) \leq c/\rho \implies OPT(I') \leq c'/\rho'$

Theorem 4.3. *Unless* $\boldsymbol{P = NP}$*, the problem of approximating TTI within a factor of* n^{ϵ}*, for any* $\epsilon > 0$ *is in* ***NP****-hard, where* n *is the number of constraint variables.*

Proof. Our starting point is the well established result that for any $\epsilon > 0$, CLIQUE is **NP**-hard to approximate within a factor of $n^{1-\epsilon}$, where n is the number of vertices in the graph [10].

The reduction from CLIQUE to Monotone-2-SAT-Max presented in the previous section is such that the optimum value for the CLIQUE instance is exactly the optimum value for the Monotone-2-SAT-Max instance. The reduction from Monotone-2-SAT-Max to TTI also preserves the optimum value. The overall reduction from CLIQUE to TTI is therefore a gap preserving reduction. For every $c, \rho > 0$, the reduction satisfies the parameters $(c, \rho), (c, \rho)$. It follows that TTI is also **NP**-hard to approximate within a factor of $n^{1-\epsilon}$, for any $\epsilon > 0$, where n is the number of constraint variables.

□

4.3 Complexity of Restricted Versions of the Problem

So far we have shown that the general TTI problem is **NP**-complete and also that it cannot be approximated efficiently to within a factor of n^{ϵ}, for some $\epsilon > 0$. We now examine the complexity of three restricted, but practical cases, of the TTI problem.

Case 1: No breakable constraints In this restricted case, $B = \emptyset$. The user can indicate preferences for certain typings by partially annotating the program with ownership modifiers. The inference is not guided by weighted heuristics. This restricted version can be efficiently solved and is therefore in **P**. If the program is unannotated, it is trivial to obtain a solution: set all constraint variables to `peer`. If the program is partially annotated, the Set Based Solution proposed for Universe Types [8] can be used to obtain a valid inference. We concluded this by first making the observation that the only kind of constraint in GUT which is not found in Universe Types is an equality constraint in which both operands are constraint variables. Such a constraint is used to handle generics. It can trivially be verified that the Set Based Solution can handle such a constraint. We also note that the Set Based Solution only works for the ranking which gives a deep ownership structure. For constraint variables which represent explicit annotations in the code this ranking is: `any` > `rep` > `peer` i.e. `any` is preferred

over `rep` which is preferred over `peer`. For constraint variables which represent results of viewpoint adaptations, this ranking is: `any` > `lost` > `rep` > `peer`. For a detailed description of the Set Based Solution the reader is referred to [8]. The same algorithm is also used in [9] to solve the inference for a reference immutability type system they propose.

Case 2: Constant number of distinct weights In this restricted case, the inference can be guided by heuristics, either built-in or provided by the user, but these must be weighted with a constant number, z, of distinct weights. In other words, the weights must be chosen from a set $\{w_1, w_2, ..., w_z\}$. This case is practical since we can select the constant to be large enough to allow sufficient expressiveness. This problem is still **NP**-complete. To prove this, we first show that Monotone-2-SAT-Max with a constant number of distinct weights is **NP**-complete. Consider the reduction from CLIQUE to Monotone-2-SAT-Max described in Sec. 4.1. The reduction reduces instances of CLIQUE to instances of Monotone-2-SAT-Max in which only one distinct weight (besides the weight of 0) is used to weigh breakable constraints i.e. $z = 1$. Hence, Monotone-2-SAT-Max with a single distinct weight is **NP**-complete. Since the case where $z = 1$ is a subcase of all the other cases where $z > 1$, we can conclude that Monotone-2-SAT-Max with a constant number of distinct weights is **NP**-complete. Consider the reduction of Monotone-2-SAT-Max to TTI described in Sec. 4.1. The same reduction can be used to reduce instances of Monotone-2-SAT-Max with a constant number of distinct weights, to TTI with a constant number of distinct weights. Hence the latter is **NP**-complete.

Case 3: At most one breakable constraint per constraint variable In this restricted case, the inference can be guided by heuristics, but for each constraint variable at most one modifier can be given a non-zero weight. This case is practical since typically a user will not weigh each of the four possibilities of each constraint variable. Doing so would be tedious and time consuming. Instead, the user usually specifies general preferences for the different types of variables. For example, a user may prefer that field types are assigned `rep`, parameter types are assigned `any` etc. The restriction here is that the user cannot specify any second or third preferences for any constraint variable. This version of the problem is still **NP**-complete. To prove this, we first show that Monotone-2-SAT-Max in which for each boolean variable, at most one assignment, either true or false, can be given a non-zero weight, is **NP**-complete. Consider the reduction from CLIQUE to Monotone-2-SAT-Max described in Sec. 4.1. The reduction reduces instances of CLIQUE to instances of Monotone-2-SAT-Max in which at most one assignment for each boolean variable is non-zero weighted. Hence, Monotone-2-SAT-Max, in which at most one assignment per boolean variable is weighted, is **NP**-complete. The reduction from Monotone-2-SAT-Max to TTI described in Sec. 4.1 can be used to

reduce this version of Monotone-2-SAT-Max to TTI in which each constraint variable is involved in at most one breakable constraint. Hence this restricted version of TTI is also **NP**-complete.

4.4 A Discussion on Identifying the Sources of Complexity

Having proved that the TTI problem for GUT is **NP**-hard, a natural question which follows is what are some of the sources of the problem's intractability? In this section we discuss some of our ideas on how to gain this insight. We also describe the roadblocks we are encountering to accomplish them at this point in time.

We first make the interesting observation that TTI in the absence of any breakable constraints is in **P**. This was proved in Sec. 4.3. Hence the breakable constraints, that is, the weighting, is a necessary contributing factor to the hardness of the problem. In other words, every hard TTI problem must be one involving weights. However, every weighted TTI problem is not hard, as is proved by the seemingly 'easy' benchmarks used in [1].

Another observation is that in the reduction from TTI to Monotone-2-SAT-Max, outlined in Sec. 4.1, the resulting program is one that has a very distinct structure. All of the classes in it form a chain of inheritance, and for each disjunction in the Monotone-2-SAT-Max formula, there is a cast in the corresponding class in the program. This structure does not seem to be one that is typically found in practical programs.

From such observations, the following questions among others arise: Besides weighting, do certain structural properties of a program, for example, depth of inheritance, also contribute to the problem's hardness? If yes, do these properties frequently occur in practical programs? If we restrict the kinds of input instances allowed, can we then devise a polynomial-time algorithm for the inference?

One idea to gain further insight into the complexity of TTI is to draw an analogy from SAT. In [15] and [16], various features of random SAT instances which influence their empirical hardness are identified and classified into groups. We can apply the same idea to the TTI problem by trying to relate these SAT features to features in TTI. For example, Group 1 in [15] relates to the problem's size and includes features such as the number of variables, number of clauses, ratio of clauses to variables, reciprocal of this ratio etc. Similarly, the most natural way to define a TTI instance's size seems to be by the number of constraint variables and constraints. As another example, Group 2 in

[15] consists of features related to the Variable-Clause Graph of the instance. This is a graph which has a node for every variable and clause and an edge between a variable node and a clause node if the variable occurs within the clause. We conjecture that in the TTI domain, the parallel of the Variable-Clause Graph is a Constraint Variable - Constraint Graph, which has a node for every constraint variable and constraint and an edge between a variable and constraint if the variable is involved in the constraint. In a similar fashion, many of the features identified for SAT in [15] which also seem relevant for TTI, can be related to some relationship between constraint variables and constraints (mandatory and breakable).

In order to verify if these TTI features have an influence on empirical hardness we require samples of random easy and hard TTI instances. Initially we felt that a possible approach to generate easy and hard TTI instances is as follows: Reduce randomly generated easy and hard SAT instances to TTI instances. We expect the resulting TTI instances to be respectively easy and hard too. A CNF-SAT instance can be reduced to TTI by going through the following chain of reductions:

CNF-SAT $<$ 3-SAT $<$ CLIQUE $<$ Monotone-2-SAT-Max $<$ TTI

Reductions from CNF-SAT to 3-SAT and from 3-SAT to CLIQUE can be found in [11]. The remaining reductions were described in Sec. 4.1. A problem with this approach is that it is doubtful whether the resulting TTI instances are really random since, owing to our reduction in Sec. 4.1, they all have the same kind of program structure, that is, a chain of inheritance and strange casts.

Besides being unable to generate random hard TTI instances, there are two other obstacles which we faced. The first one is that the GUT inference tool is currently not in fully working condition. In 2011, the tool was implemented on top of the Checker Framework, a pluggable type checking framework built on top of the JSR 308 branch of the OpenJDK compiler [1]. The Checker Framework provides an abstraction of a basic type checker. As a result, building the inference tool on top of it significantly simplified the implementation of many language features. Since then however, the Checker Framework has gone through many changes and the inference tool has not been updated to reflect these.

Hoping to bypass the tool, we reduced the TTI instances directly to partial weighted Max-SAT instances using the reduction outlined in Sec. 3.2. We then ran a partial weighted Max-SAT solver, SAT4J, on them in order to confirm if they remained easy and hard respectively in the TTI domain. We found that for many easy SAT instances, the resulting TTI instances when reduced to partial weighted Max-SAT could not be solved efficiently with SAT4J. This implies that either (1) The instance is an easy

TTI instance but too complicated for SAT4J, or (2) The instance is an inherently difficult TTI instance. From performance evaluations conducted on different partial weighted Max-SAT solvers, we find that SAT4J is among those that performs poorly: there are more than a handful of solvers which perform significantly better than it [13]. Unfortunately, the more efficient SAT solvers are currently not publicly available for use.

Chapter 5

Conclusions and Future Work

In this thesis, we analyzed the computational complexity of the Tunable Type Inference (TTI) problem for Generic Universe Types, an ownership-based type system. This is the problem of assigning ownership modifiers to constraint variables, ensuring that the assignment satisfies all mandatory type constraints and achieves the maximum possible weight from among breakable, preference constraints.

We proved that the corresponding decision version of the TTI problem is **NP**-complete. The proof went as follows: We first introduced a new problem called Monotone-2-SAT-Max, proved that the new problem is **NP**-complete by a reduction from CLIQUE, and then reduced the new problem to TTI thereby proving that the latter is also **NP**-complete.

Using a gap-preserving reduction, we then proved that not only is the TTI problem hard to solve exactly, it is also hard to approximate to within a certain factor. To do this, we made use of the fact that in the reductions from CLIQUE to Monotone-2-SAT-Max and from Monotone-2-SAT-Max to TTI, the optimal values are preserved.

Three restricted but practical versions of TTI were then analyzed. In the first version, preferences could only be expressed by partial annotations to the source code, and there were no breakable constraints. This case was found to be in **P** with a possible solution being the Set Based Solution proposed in [8]. In the second version, only a constant number of distinct weights could be used to express preferences. This case was found to be in **NP**-complete. In fact, we showed that even if only 1 distinct weight is used (besides the default weight of 0), the problem is still in **NP**-complete. In the third version, each variable could be involved in at most one breakable constraint. This case was also found to be in **NP**-complete.

Although we presented some ideas on identifying the sources of complexity of TTI, we were unable to draw any final conclusions about it. Whether or not program structure influences the empirical hardness of TTI is still an open question.

Our reduction from Monotone-2-SAT-Max to TTI makes use of the comparable constraint. However this kind of constraint is used only for casts. There is an alternative to dealing with casts: the static inference can choose modifiers in such a way as to guarantee that the runtime check at each cast succeeds. This is accomplished by changing "<:>" to "=" [1]. An interesting question is whether TTI remains **NP**-complete if this alternative approach is adopted.

Furthermore, it would be interesting to know if the problem's complexity changes if the owner-as-modifier option is forbidden, since our reduction did make use of this encapsulation scheme.

As a first step to answering these questions, we propose having the inference tool updated and working. We also propose that the inference tool either makes use of a partial weighted Max-SAT solver which is more efficient than the SAT4J solver it currently uses, or that other solution approaches, for example integer programming, are explored.

References

[1] W. Dietl, M. D. Ernst and P. Müller, *Tunable Static Inference for Generic Universe Types*, European Conference on Object-Oriented Programming (ECOOP), July 2011, Best Paper Award.

[2] W. Dietl, S. Drossopoulou, and P. Müller, *Generic Universe Types*, European Conference on Object-Oriented Programming (ECOOP), July 2007, pp. 28-53.

[3] W. Dietl, *Universe Types: Topology, Encapsulation, Genericity, and Tools*, PhD thesis, Department of Computer Science, ETH Zurich, 2009.

[4] C. Flanagan and S. N. Freund, *Type Inference Against Races*, Science of Computer Programming, Vol 64, Issue 1, January 2007, pp. 140-165.

[5] M. Elder and B. Liblit, *Heap Typability Is NP-Complete*, 2007.

[6] J. Aldrich, V. Kostadinov, and C. Chambers, *Alias Annotations for Program Understanding*, Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), 2002, pp 311-330.

[7] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg, *Inference of User-defined Type-qualifier Rules*, European Symposium on Programming (ESOP), Vol 3924, pp. 264-278, 2006.

[8] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst, *Inference and Checking of Object Ownership*, European Conference on Object-Oriented Programming (ECOOP), June 2012.

[9] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, *ReIm & ReImInfer: Checking and inference of reference immutability and method purity*, Object-Oriented Programming, Systems, Languages & Applications, October 2012.

[10] J. Hastad, *Clique is Hard to Approximate within* $n^{1-\epsilon}$, Foundations of Computer Science, 1996.

[11] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, Third Ed, 2009, Ch. 34.

[12] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre, *A simple model to generate hard satisfiable instances*, Proceedings of the 19th International Joint Conference on Artifical Intelligence, 2005.

[13] Max-SAT 2014, *http://www.maxsat.udl.cat/14/results/index.html*, 9th Max-SAT Evaluation Results.

[14] M. Papi, M. Ali, T. Correa Jr., J. Perkins and M. D. Ernst, *Practical Pluggable Types for Java*, ISSTA, pp. 201-212, 2008.

[15] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, *Understanding Random SAT: Beyond the Clauses-to-Variables Ratio*, Proceedings of Principles and Practice of Constraint Programming, 2004.

[16] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, *SATzilla: Portfoliobased Algorithm Selection for SAT*, Journal of Artificial Intelligence Research, 2008.

[17] A. Igarashi , B. C. Pierce and P. Wadler, *Featherweight Java: A Minimal Core Calculus for Java and GJ*, ACM Transactions on Programming Languages and Systems, 1999.