WITNESS FUNCTIONS IN PROGRAM ANALYSIS AND COMPLEXITY THEORY

A Dissertation
Presented to
The Academic Faculty

By

Shuo Ding

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

Dec 2024

WITNESS FUNCTIONS IN PROGRAM ANALYSIS AND COMPLEXITY THEORY

Thesis committee:

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Jens Palsberg
Computer Science Department
University of California, Los Angeles

Dr. Suguman Bansal
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Vijay Ganesh
School of Computer Science
Georgia Institute of Technology

Date approved: December, 2024

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Qirun Zhang for his help and guidance through my PhD journey. In particular, his support and encouragement on my explorations on logic and computability theory is one of the key reason why this thesis can be formed.

I would also like to thank other committee members, Dr. Suguman Bansal, Dr. Vijay Ganesh, Dr. Jens Palsberg, Dr. Vivek Sarkar, for agreeing to spend time and provide valuable feedback on my research.

I enjoyed my two internships, one at Amazon and another one at Meta, during my PhD study. The internship projects broadened my view on industrial needs and inspired some of my practical research directions.

I thank my friends that I met in US and China for their help and support. They also provided many unique perspectives to the research problems that I encountered, and stimulated my interest in interdisciplinary areas.

Finally, I wish to extend my gratitude to my family for their continuous support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Proving impossibility results is one of the main themes of program analysis theory and computability/complexity theory. For example, we can prove a program analysis problem is undecidable, meaning that there does not exist an algorithm to precisely solve the problem. As another example, we can prove a problem does not belong to a complexity class, meaning that every correct algorithm for the problem must exceed the given resource restriction. In general, given a class $\mathbf{C}$ of computational problems and a specific computational problem P not in $\mathbf{C}$, a witness function maps every candidate Q in $\mathbf{C}$ to an input on which P and Q are different.

We investigate the computational properties of such witness functions and discuss their implications. In program analysis theory, we prove that a large class of undecidable program analysis problems have computable witness functions, including every semantic property described in Rice's theorem. This implies the existence of computable functions mapping every program analyzer to a more precise program analyzer. Through two real program analysis tasks (1) CFL-reachability based program analysis for Java and LLVM-IR and (2) template constraint analysis for C++, we demonstrate that computable witness functions provide guarantees on the progress of developing more and more precise program analysis techniques. In complexity theory, we prove that witness functions for major complexity classes are closely related to reductions, and discuss the implications in complexity class separation proofs.

CHAPTER 1

INTRODUCTION

Computability theory and complexity theory [1, 2] study the hardness of computational problems. One of the main themes in these areas is to prove impossibility results, e.g. a problem is undecidable, is complete in a complexity class, or does not belong to a complexity class. Program analysis and related areas, which are trying to automatically determine semantic properties of computer programs, often deal with undecidable problems [3, 4, 5, 6] and thus is closely related to computability theory.

However, there are still missing pieces of understanding computability-theoretic foundations for program analysis and complexity-theoretic separation proofs. First, in program analysis, undecidability [7, 8] and degree structures (*e.g.*, many-one degrees, and Turing degrees [1]) only tell us the impossibility of solving program analysis problems precisely, but does not shed light on improving program analysis techniques, which are essentially decidable approximations of undecidable problems [9, 10, 11]. Second, in complexity-theoretic separation proofs, the known barrier results, such as the relativization barrier [12] showing relativizing proofs cannot resolve the $\mathbf{P}$ versus $\mathbf{NP}$ problem, demonstrate what a separation proof cannot look like but do not demonstrate what a separation proof must look like.

We use the concept called *witness functions* to study more about these impossibility scenarios. Given a class $\mathbf{C}$ of computational problems and a specific computational problem $P \notin \mathbf{C}$, a witness function maps every candidate $Q \in \mathbf{C}$ to an input on which P and Q are different. It is straightforward to see that the existence of such witness functions is equivalent to the separation $P \notin \mathbf{C}$. Witness functions [13, 14] or similar concepts (such as productive functions [15]) has been considered in previous work for other purposes.

Specifically, we investigate the computational properties of witness functions in two

settings: (1) the undecidability of program analysis problems P with respect to the class of computable functions $\mathbf{C}$, which corresponds to program analysis or verification algorithms, and (2) the separation of computable problems P from complexity classes $\mathbf{C}$. In the first case, we prove that many program analysis problems, including all the semantic properties described in Rice's theorem [16], admit *computable* witness functions, which implies the existence of computable transformations converting any program analyzer to a more precise program analyzer. In the second case, we prove that the complexity of witness functions for complexity class separations is closely related to a specific type of reductions, called "universal reductions", and deduce properties of such witness functions. This eventually implies that any correct complexity theoretic separation proof, which asserts the existence of witness functions, essentially transforms the lower-bounded problem P to a form close to the artificially constructed problems in time / space hierarchy theorems [17, 18].

On the practical side, we consider two program analysis examples: (1) mutual refinement of CFL-reachability-based program analysis for Java and LLVM-IR and (2) template constraint analysis for C++. In both cases, the exact problems are undecidable [4, 19]. We demonstrate that the existence of computable witness functions provide formal guarantees on the progress of improving such practical program analysis techniques. In the two program analysis examples, both of our methods (mutual refinement and constraint synthesis) can serve as total computable approximations to undecidable problems, and thus computable witness functions can take the implementations of these two methods and generate corresponding counterexamples, from which more precise techniques can be derived. In practice, precision improvements on these program analysis techniques are often obtained by domain-specific knowledge and ad-hoc reasoning, but these efforts may not succeed. Our computable witness functions, on the other hand, provide computable improvements that are guaranteed to succeed.

Regarding the practical implications on program analysis, another important detail is that computable witness functions work on decision problems, so we need to first transform

the program analysis problem to a decision problem (with Yes/No answers) and then apply the witness function. For mutual refinement, a decision problem version asks whether a specific pair of nodes in the graph is a reachable pair. For template constraint synthesis, a decision problem version asks whether a specific template parameter is constrained by a specific predicate.

1.1 Thesis Statement

Properties about computability/complexity theoretic witness functions provide theoretical guarantees on the progress of developing more and more precise program reasoning algorithms and theoretical implications on complexity class separation proofs.

1.2 Thesis Contributions

In this section, we summarize the four pieces of our work about witness functions, which correspond to Chapter 3-Chapter 6.

1.2.1 Witnessability of Undecidable Problems

Many problems in programming language theory and formal methods are undecidable, and practical techniques dealing with these problems are often based on decidable approximations and are always imprecise. Typically, practitioners use heuristics and *ad hoc* reasoning to identify imprecision issues and improve approximations, but there is a lack of computability-theoretic foundations about whether those efforts can succeed.

This chapter shows a surprising interplay between undecidability and decidable approximations: there exists a class of undecidable problems, such that it is computable to transform any decidable approximation to a witness input demonstrating its imprecision. We call those undecidable problems *witnessable problems*. For example, if a program property P is witnessable, then there exists a computable function f_P, such that f_P takes as input the code of any program analyzer targeting P and produces an input program w

on which the program analyzer is imprecise. We prove the diagonal halting problem K is witnessable, and the class of witnessable problems is closed under complements and many-one reductions. In particular, all "non-trivial semantic properties of programs" mentioned in Rice's theorem are witnessable. We also explicitly construct a problem in the non-witnessable (and undecidable) class and show that both classes have cardinality $2^{\aleph_0}$. These results offer a new perspective on the understanding of undecidability: for witnessable problems, although it is impossible to solve them precisely, it is always possible to improve any decidable approximation to make it closer to the precise solution. This fact formally demonstrates that research efforts on such approximations are promising.

1.2.2 On Witness Functions for Complexity Lower Bounds

For a (possibly hypothetical) complexity theoretic separation such as $\mathrm{SAT} \notin \mathbf{P}$, consider a "witness function" w transforming any program in $\mathbf{P}$ to an input on which it is different from SAT. The witness function can be regarded as a constructive form of separation. This chapter discusses detailed properties of such witness functions, including their close relations with reductions, and shows three implications: (1) constant additive/multiplicative factors in complexity class definitions are important in constructive separation proofs, (2) any successful separation of $\mathrm{SAT} \notin \mathbf{P}$ implicitly contains an effort to transform SAT to an artificial problem similar to the ones in hierarchy theorems, and (3) witness functions exist in real complexity theoretic proofs. The results are presented in an axiomatic way so they are mostly machine-independent and complexity-class independent.

1.2.3 Example: Mutual Refinements of Context-Free Language Reachability

Context-free language reachability is an important program analysis framework, but the exact analysis problems can be intractable or undecidable, where CFL-reachability approximates such problems. For the same problem, there could be many over-approximations based on different CFLs $C_1, \ldots, C_n$. Suppose the reachability result of each C_i produces a

set P_i of reachable vertex pairs. Is it possible to achieve better precision than the straightforward intersection $\bigcap_{i=1}^{n} P_i$?

This chapter gives an affirmative answer: although CFLs are not closed under intersections, in CFL-reachability we can "intersect" graphs. Specifically, we propose *mutual refinement* to combine different CFL-reachability-based over-approximations. Our key insight is that the standard CFL-reachability algorithm can be slightly modified to trace the edges that contribute to the reachability results of C_1, and C_2-reachability only need to consider contributing edges of C_1, which can, in turn, trace the edges that contribute to C_2-reachability, etc. We prove that there exists a unique optimal refinement result (fix-point). Experimental results show that mutual refinement can achieve better precision than the straightforward intersection with reasonable extra cost.

Regarding each CFL as a decidable approximation, this refinement process gives an example of improving such decidable approximations. Furthermore, according to the witnessability result, as long as the undecidable problem is witnessable, there exist more precise approximations that can be computably obtained even after the fix-point is reached, which makes a guarantee of success for research efforts aiming to further improve the precision of the mutual refinement method. In particular, in this specific scenario, the witness function takes the implementation of mutual refinement as input, and produces a graph together with a pair of vertices which is unreachable, but mutual refinement concludes that it is a reachable pair.

1.2.4 Example: Fast Constraint Synthesis for `C++` Function Templates

`C++` templates are a powerful feature for generic programming and compile-time computations. It is well known that `C++` compilers often emit overly verbose template error messages, and even short error messages often involve unnecessary and confusing implementation details, which are difficult for developers to read and understand. To address this problem, `C++20` introduced *constraints and concepts*, which impose requirements on

template parameters. The new features can define clearer interfaces for templates and can improve the compiler diagnostics for failed template instantiations. However, manually specifying template constraints can still be a non-trivial task, and precise constraint inference is undecidable due to C++ templates' flexibility. This task becomes even more challenging when working with legacy C++ projects or with frequent code changes during development.

This chapter bridges the gap and proposes the first automatic approach to synthesizing constraints for C++ function templates. Our approach utilizes a lightweight static analysis to analyze the usage patterns within the template body and summarizes them into constraints for each type parameter of the template. The analysis is inter-procedural and uses disjunctions of constraints to model function overloading. We have implemented our approach based on the Clang frontend and evaluated it on two C++ libraries chosen separately from two popular library sets: `algorithm` from the Standard Template Library (STL) and `special_functions` from the Boost library, both of which extensively use templates. Our tool can process over 110k lines of C++ code in less than 1.5 seconds and synthesize non-trivial constraints for 30%-40% of the function templates. Furthermore, the constraints synthesized for `algorithm` align well with the standard documentation, and on average, the synthesized constraints can reduce error message lengths by 56.6% for `algorithm` and 63.8% for `special_functions`.

Because of the undecidability and witnessability of precise constraint synthesis, our lightweight method, which is a decidable approximation, can be computably transformed into a more precise approximation. Thus, properties about witness functions can also provide guarantees on this seemingly different scenario of program analysis. In particular, the witness function takes the implementation of the constraint synthesis algorithm as input, and produces a C++ program together with a constraint on a template parameter, such that the parameter does not require that constraint in reality, but the constraint synthesis algorithm concludes that it requires that constraint.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents basic concepts and notations used throughout later chapters. Chapter 3 presents the existence of computable witness functions in a large class of undecidable problems. Chapter 4 studies the witness functions in the complexity theory setting and proposes implications on complexity class separation proofs. Chapter 5 presents our first program analysis scenario of formal language reachability. Chapter 6 presents our second program analysis scenario of C++ template constraint inference. Finally, Chapter 7 concludes the thesis and presents some future directions.

CHAPTER 2

PRELIMINARY

This chapter presents an overall preliminary for the entire document, which includes the most fundamental concepts and definitions used throughout later chapters. Examples are provided to illustrate these basic concepts.

2.1 Logic Background

2.1.1 Set Theory

We adopt the standard set theory notations [20]: the "belongs to" relation $\in$, the strict subset relation $\subset$, the non-strict subset relation $\subseteq$, the set difference operator $\setminus$, and the set symmetric difference operator $\triangle$. In particular, $\triangle$ is defined as follows.

$$A \triangle B = (A \setminus B) \cup (B \setminus A).$$

In addition to that, we use $\mathbb{N}$ to denote the set of all natural numbers. Upper case letters $X, Y, Z \cdots$ denote subsets of $\mathbb{N}$, and lower case letters $x, y, z \cdots$ denote natural numbers. Given a set $S \subseteq \mathbb{N}$, the complement of S with respect to $\mathbb{N}$ is denoted as S^c, the power set of S is denoted as $\mathcal{P}(S)$, and the characteristic function χ_S of S is a function from $\mathbb{N}$ to the set $\{0, 1\}$ defined as follows:

$$\chi_S(x) = \begin{cases} 1 & \text{if} \quad x \in S \\ 0 & \text{if} \quad x \notin S. \end{cases}$$

We use boldface letters $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ to represent sets of subsets of $\mathbb{N}$, or equivalently, subsets of $\mathcal{P}(\mathbb{N})$.

2.1.2 Gödel's Encoding

Two common ways to formulate computability theory and complexity theory are formal languages [2] and sets of natural numbers [1, 21]. We follow the second approach: studying the (possibly relative) computability of subsets of $\mathbb{N}$ and partial functions from $\mathbb{N}$ to $\mathbb{N}$.

Gödel's encoding [22, 23] is a technique used to encode finite objects (e.g. strings, trees, etc.) as natural numbers, which is the rationale behind using natural numbers to discuss computability/complexity. For example, given an alphabet $\Sigma = \{a, b, c\}$, one way to encode strings in Σ^* into natural numbers is to first assign a fixed natural number to every character in the alphabet.

$$a \mapsto 1,\ b \mapsto 2,\ c \mapsto 3$$

Then we can utilize the infinite list of prime numbers to encode a finite string such as "aacbb" into products of prime number powers:

$$2^1 3^1 5^3 7^2 11^2.$$

By the fundamental theorem of arithmetic [24], every integer greater than 1 can be represented uniquely as a product of prime numbers, and thus the decoding process is also computable.

Another way to encode finite objects as natural numbers is to fix an ordered enumeration of such objects, and then use the index in the list as the encoding of the corresponding object. For example, we can list all binary strings in the following order.

$$\epsilon, 0, 00, 01, 10, 11, \ldots$$

Then the natural number encoding of 00 is 2, because its index is 2 in the above enumeration. The encoding and decoding of this situation is still computable, because we can

enumerate the list until we find the correspondence.

In general, in computability settings, the specific ways of encoding is flexible as long as both the encoding and the decoding processes are computable.

2.1.3 Numbering of Programs

We will use the notation $\phi_i(x)$ frequently in Chapter 3 and Chapter 4, which refers to program i's value on input x. Both i and x are natural numbers, and other data types can be encoded as natural numbers (or equivalently, the binary representations of natural numbers). In particular, multi-input programs can be represented as single-input programs via encoding of pairs and tuples. In this way we can enumerate all partial computable functions $\{\phi_0, \phi_1, \ldots\}$. The notation "$\phi_i(x) \downarrow$" means that ϕ_i is defined on x and the notation "$\phi_i(x) \uparrow$" means that ϕ_i is undefined (or divergent) on x. As an analogy, the undefined case of partial computable functions corresponds to the error state or the non-termination state of computer programs on a specific input. A partial computable function is total if and only if it is defined on every input.

Note that there are infinitely many different indices corresponding to any single partial computable function, resembling the fact that there are infinitely many ways to implement the same function in common programming languages. This fact holds for all admissible numberings according to Rogers' equivalence theorem [25]: in fact, there exist computable bijections between any two admissible numberings. A numbering is admissible (which is also called acceptable [15]) if and only if it satisfies the following conditions.

- The universal function $u(e, x) = \phi_e(x)$ is a partial computable function.
- $\{\phi_i : i \in \mathbb{N}\}$ includes all partial computable functions.
- The S-m-n theorem holds: there exists a total computable function s such that $\forall e, x, y \in \mathbb{N}, \phi_{s(e,x)}(y) = \phi_e(x, y)$.

2.1.4 Type Systems and Natural Number Encoding

In real programming languages, type systems [26, 27, 11] are often employed to classify data types, so a program's inputs do not need to be natural numbers. Under specific definitions of programming language concepts such as interpreters, the presence of types systems can affect the computability/complexity theoretic conclusions: for example, the normalization barrier does not hold under typed representations of inputs [28].

However, using Gödel's encoding or similar techniques we can always encode typed inputs as natural numbers, and in this case the type system's effects can be reflected using type error messages, which can be encoded as natural numbers as well. In particular, real-world typed language interpreters like Haskell GHCi [29] does type-checking inside their interpreters, emitting type errors (which are essentially special values printed) on ill-typed input programs. Since programs are eventually represented as strings (and thus can be encoded as natural numbers), type checkers built inside interpreters are the common practice. Under this setting where everything has an untyped representation, the normalization barrier can be recovered [30]. In summary, in computability settings, it is often sufficient to only consider natural number inputs, and the type system's effects can be embedded into natural numbers as well.

2.2 Programming Language Background

2.2.1 L-Reachability

In program analysis, the L-reachability problem is to find pairs (s, t) of vertices such that there exists a path from s to t, and the edge labels along that path form a string in L.

Definition 1 (L-Reachability)**.** *Given a formal language L with a finite alphabet Σ and a finite graph $G = (V, E)$, where each edge $e \in E$ is labeled with a character in Σ, vertex $t \in V$ is L-reachable from vertex $s \in V$ if and only if there exists a finite path (with possibly duplicate vertices and edges) $p = s \xrightarrow{l_1} v_1 \xrightarrow{l_2} \ldots \xrightarrow{l_{n-1}} v_{n-1} \xrightarrow{l_n} t$ in the graph such that the*

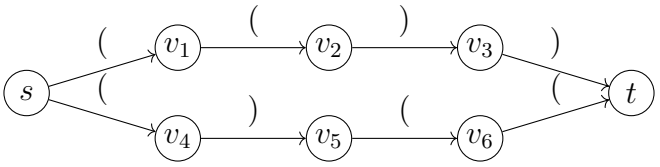


Figure 2.1: An example of Dyck-reachability.

string $l_1 l_2 \dots l_n$ is in the given formal language. $R(p) = l_1 l_2 \dots l_n$ is the path string of p. The zero-length path from a vertex to itself forms the empty string. (s, t) is an L-reachable pair. The L-reachability problem is to find the set of all L-reachable pairs.

Example 1. *Consider the following example of Dyck-reachability, where the Dyck language is specified by the following context-free grammar D.*

$$D \ \rightarrow D\ D \mid (\ D\ ) \mid \epsilon.$$

Then the edge-labeled graph shown in Figure 2.1 is an instance of Dyck-reachability. In this specific case, (s, t) is a D-reachable pair because the path $s \xrightarrow{(} v_1 \xrightarrow{(} v_2 \xrightarrow{)} v_3 \xrightarrow{)} t$ forms a string in D.

Fix a formal language L. Any graph $G = (V, E)$ whose edges are labeled with characters in the alphabet of L essentially gives an instance of the L-reachability problem. We denote this instance as $\langle L, (V, E) \rangle$. There also exist other variants of L-reachability, such as the single-pair reachability, which only cares about the reachability between a specific pair of vertices. In Chapter 5, we focus on all-pairs reachability unless otherwise noted.

2.2.2 The C++ Template System

In C++, a *function template* F defines a family of functions. Abstractly, F takes a list of template arguments $\vec{a}$ and returns a concrete C++ function $F(\vec{a})$; this computation, normally called template instantiation, is done in compile-time. Since template arguments $\vec{a}$ might result in type errors during the instantiation of function template F, a *constraint* can be associated with F to specify requirements on F's template arguments. A named set

```cpp
#include <concepts>

template <std::integral T>
T three(T x) {
    return x + x + x;
}

int main() {}
```

Figure 2.2: An example illustrating C++ templates and concepts.

of such constraints is called a *concept*. A concept C is a compile-time predicate taking a list of template arguments $\vec{a}$ and returning `true` or `false`, and C can be used to specify requirements for multiple function templates.

Example 2. *Figure 2.2 illustrates the basic mechanism of C++ templates and constraints/-concepts. The function template `three` has one template argument `T`, which is constrained by the built-in concept `std::integral`. Constraints and concepts were introduced in C++20. The constraint specifies that T must be integer types, which can detect erroneous instantiations early and emit short and easy-to-understand error messages.*

CHAPTER 3

WITNESSABILITY OF UNDECIDABLE PROBLEMS

3.1 Introduction

Many problems in programming language theory and formal methods (program analysis [3, 4], program verification [5, 6], SMT solving [31, 32], type systems [26, 27, 11], etc.) consider complicated objects such as programs written in Turing-complete languages, and those problems are proved to be undecidable. It has been well-known since Turing and Church [7, 8] that undecidable problems cannot be solved precisely. In practice, perhaps the best-known technique for handling undecidable problems is utilizing decidable approximations [9, 10, 11]. Undecidability implies that all approximations must be imprecise on infinitely many inputs—although theoretically important, it is a relatively discouraging result, and it does not shed light on improving the approximations encountered in practice.

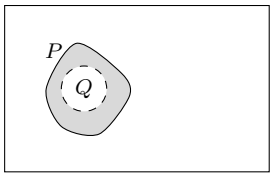
This chapter goes beyond previous work by presenting a surprising interplay between undecidability and decidable approximations. Specifically, we show that for a large class of undecidable problems, there exist *computable functions* that take as input the implementation (source code) of a decidable approximation and output a witness on which the approximation is imprecise. At first glance, this result appears counter-intuitive because, due to the nature of undecidability, for any arbitrarily given input, there is no general way to tell whether the approximation is imprecise on it. Otherwise, the problem would be decidable. Our result shows that there exists an algorithm that can compute imprecise inputs from the approximations. Our result does not aim to decide whether the approximations are imprecise on arbitrary inputs, thus bypassing the undecidability. Furthermore, this enables an iterative process: after computing a witness and improving the existing approximation, our result shows that we can always obtain a new witness, *i.e.*, the iterative process leads to

more and more precise approximations. Note that this is fundamentally different from the idea of CounterExample-Guided Abstraction Refinement (CEGAR) [33]: (1) in program verification, CEGAR often refines abstractions for each input program while we refine the program verifier itself; and (2) even if we apply the idea of CEGAR to refine a sound[1] program verifier when the verifier fails to prove the correctness of a program, it is, in general, undecidable to know whether that is a false positive. On the contrary, our approach directly constructs correct programs that cannot be proved correct by the verifier.

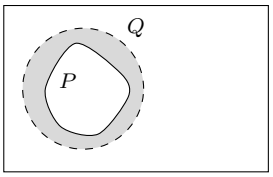
We state and prove our results using the terminology of computability theory. In the literature, computability theory has been primarily discussed using formal languages [2] and sets of natural numbers [1, 21], and the two approaches are equivalent. Our work adopts the second approach: undecidable problems and their decidable approximations are both modeled as sets of natural numbers. Our result is general and applies to any Turing-complete programming language. In particular, natural numbers can encode any finite amount of information by computable encodings [22, 23]. Consider an undecidable problem P and its decidable approximations Q in Figure 3.1. Different approximation abstractions can lead to several set relationships between P and Q: Q is a subset of P (Figure 3.1a), Q is a superset of P (Figure 3.1b), Q intersects with P but is neither a subset nor a superset of P (Figure 3.1c), Q is disjoint from P (Figure 3.1d). To discuss all cases uniformly, we call the *symmetric difference* $P \triangle Q = (P \setminus Q) \cup (Q \setminus P)$ the *imprecision* of the approximation. For example, if Q is an under-approximation of P, then $P \triangle Q = P \setminus Q$, which represents the area of P not covered by Q. Utilizing the symmetric difference $P \triangle Q$ makes our results more general because the approximations are not restricted to under-approximations (Figure 3.1a) or over-approximations (Figure 3.1b).

We say an undecidable problem P is *witnessable* if and only if there exists a partial computable function w_P only depending on P, such that for any decidable approximation

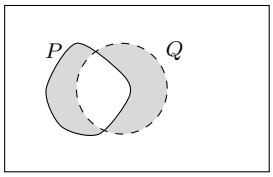
[1] "Sound" means if the verifier concludes the program is correct, then the program is indeed correct. In other words, the verifier forms an *under-approximation* of the set of correct programs, which is typically implemented by *over-approximating* programs' behaviors (thus rejecting some correct programs). This convention is used throughout chapter.



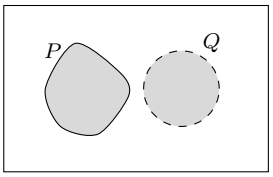
(a) Under-approximation.



(b) Over-approximation.



(c) Non-disjoint.



(d) Disjoint.

Figure 3.1: Four cases of an undecidable problem P and its decidable approximation Q. The grey areas ($P \triangle Q$) represent the imprecision. The trivial case of $Q = \emptyset$ could be classified as either case (a) or case (d). The rectangle surrounding each case represents the set of all natural numbers.

Q and its characteristic function[2] ϕ_q (the program implementing ϕ_q is encoded as a natural number q), $w_P(q)$ is defined (denoted as $w_P(q) \downarrow$) and $w_P(q) \in P \triangle Q$. Therefore, $w_P(q)$ is an *imprecision witness* of Q and w_P is a *witness function* of P. Our definition resembles the definition of productive sets in computability theory [16], but we focus on decidable approximations and do not require the approximation Q to be a subset of P.

This chapter proves the following main results.

1. The diagonal halting problem[3] $K = \{i \mid \phi_i(i) \downarrow\}$ is witnessable (Theorem 2);
2. If P is witnessable, then its complement P^c is also witnessable (Theorem 3);
3. If P_1 is witnessable and P_1 is many-one reducible to P_2, P_2 is also witnessable (Theorem 4).

These facts show that witnessable problems cover many undecidable problems in programming language theory and related fields. In particular, all "non-trivial semantics properties of programs" mentioned in Rice's theorem [1] and all "non-trivial complexity

[2]Recall that a set S's characteristic function is a 0-1 valued function f such that $f(x) = 1 \Leftrightarrow x \in S$.

[3]The diagonal halting problem and the traditional form of halting problem [2] are many-one reducible to each other.

cliques" mentioned in intensional Rice's theorem [21] are witnessable. The satisfiability and validity of first-order logic formulas [7] and the Post correspondence problem [34] are also witnessable. Witnessability cannot be achieved via simple enumeration: in Figure 3.1b, by naively enumerating all programs and checking each of them using the characteristic function of Q, we are only guaranteed to find programs in Q or Q^c. But for programs in Q, we may never know whether they are in the symmetric difference area $P \triangle Q$ (marked as grey in Figure 3.1) because P is undecidable. Our approach, however, can directly compute a program that belongs to $P \triangle Q$.

The implications of our result are threefold.

1. It shows the existence of universal ways to identify the precision issues of many algorithms in programming language theory and formal methods, including but not limited to program analyzers, program verifiers, SMT solvers, etc. The only restriction is that the algorithms should be total (*i.e.*, they terminate on every input).

2. It shows common undecidable problems encountered in programming language theory and formal methods are more "tangible" than the folklore intuition of "being impossible to solve." In particular, although they are undecidable, the process of improving any given decidable approximation is computable. This provides a theoretical foundation for justifying why research efforts targeting those problems are promising and work well in practice.

3. Many mathematical methods used in undecidability proofs are commonly regarded as ways to prove negative results (*e.g.*, undecidability). However, our results show that they also give ways to improve any given decidable approximation (thus, they also have positive effects).

The rest of chapter is organized as follows. Section 3.2 explains our basic notations and reviews computability theory. Section 3.3 gives our main results about witnessable problems. Section 3.4 explicitly constructs a non-witnessable problem. Section 3.5 shows

the cardinalities of the two classes of problems. Section 3.6 uses two examples (program analyzers and SMT solvers) to discuss the implications for programming language theory and related fields. Section 3.7 presents discussions. Section 3.8 surveys related work. Section 3.9 concludes and discusses future research directions.

3.2 Preliminary

Section 3.2.1 reviews computability theory, and Section 3.2.2 introduces our definition of decidable approximations.

3.2.1 Computability Theory

Partial Computable Functions We use the standard notion of k-ary partial computable functions (from $\mathbb{N}^k$ to $\mathbb{N}$, $1 \leq k < +\infty$), which are partial functions computable by Turing machines, lambda calculus, or any equivalent models of computation [2, 1]. The default arity of a partial computable function is one if it is not specified, and we mainly focus on 1-ary partial computable functions in chapter. In general, discussing 1-ary functions in computability theory suffices because a k-tuple of natural numbers can be computably converted to a single natural number and also be computably converted back (*e.g.*, Gödel's encoding [22]). The domain of a partial computable function ϕ is the set of inputs on which ϕ is defined: $\{\vec{x} \,|\, \phi(\vec{x}) \downarrow\}$. If two partial computable functions f and g have the same domain and output the same value on every input from the domain, then $f = g$ because they are the same partial computable function.

Numberings For each arity k, we fix an admissible numbering [25] (enumeration) of k-ary partial computable functions (*e.g.*, the one generated by Kleene's normal form theorem [1], or the one corresponding to an enumeration of all Turing machines):

$$\phi_0^k, \phi_1^k, \phi_2^k, \cdots,$$

where each $\phi_i^k, i \in \mathbb{N}$ is a partial computable function from $\mathbb{N}^k$ to $\mathbb{N}$. If $k = 1$ we write ϕ_i, $i \in \mathbb{N}$. As introduced in Section 2.1.3, an index i in the numbering is analogous to a "program (code)" implementing the corresponding partial computable function, and $\Phi(i, x) = \phi_i(x)$ is analogous to an interpreter for the corresponding programming language.

Computational Problems A (computational) *problem* is formulated as a subset of $\mathbb{N}$. For example, the diagonal halting problem $K = \{i \mid \phi_i(i) \downarrow\}$ is the set of natural numbers representing programs that halt on themselves. The two names "problem" and "set" (of natural numbers) are used interchangeably throughout this chapter. A problem is decidable if and only if its characteristic function is a total computable function, where any index for the characteristic function is called a *decider* for the problem. Otherwise, the problem is undecidable.

The following two facts are used in the proofs of our main results.

Fact 1. *Decidable sets are closed under complements, finite unions, finite intersections, and addition/removal of finitely many natural numbers. In particular, any finite set is decidable.*

Proof. Those operations can easily be implemented using any Turing-complete programming language. By the Church-Turing thesis, the proof is completed. □

Fact 2. *If P is an undecidable problem, then both P and P^c are infinite.*

Proof. If P or P^c is finite, then P is decidable by Fact 1, which is a contradiction. □

We use the standard definition [1] of computably enumerable (c.e.) problems (also called recursively enumerable (r.e.) problems). A problem is c.e. if and only if it is the domain of a partial computable function. A problem is co-c.e. if and only if its complement is c.e.

Many-One Reductions We use many-one reductions [1] to propagate imprecision witnesses among different problems.

Definition 2 (Many-One Reductions)**.** *A problem P is many-one reducible to another problem Q (written as $P \leq_m Q$) if and only if there exists a total computable function f such that the following holds. f is called a many-one reduction from P to Q.*

$$\forall x \in \mathbb{N}, x \in P \Leftrightarrow f(x) \in Q.$$

Intuitively, a set P is many-one reducible to a set Q shows that Q is harder than P. In the above definition, to decide whether a natural number x belongs to P, we can first apply f on x and then test whether $f(x)$ belongs to Q. Thus if we can decide Q, we can also decide P.

S-M-N Theorem We extensively use the S-m-n theorem [1] in our proofs. This theorem is analogous to partial evaluation [35] in programming languages.

Theorem 1 (S-m-n)**.** *For any $m, n \in \mathbb{N}$, there exists an $(m+1)$-ary total computable function ψ_n^m such that the following holds for all $i, x_1, \cdots, x_m, y_1, \cdots, y_n \in \mathbb{N}$.*

$$\phi^n_{\psi_n^m(i,x_1,\cdots,x_m)}(y_1, \cdots, y_n) = \phi_i^{m+n}(x_1, \cdots, x_m, y_1, \cdots, y_n).$$

ψ_n^m roughly corresponds to partial evaluators in programming languages. A trivial implementation of ψ_n^m is wrapping the code of ϕ_i^{m+n} (which is i) using fixed values of the first m inputs.

3.2.2 Decidable Approximations

Decidable approximations are commonly used in program analysis, program verification, etc., to deal with undecidable problems. Technically, for an undecidable set P, any decidable set of natural numbers could be regarded as an approximation of P, and different approximations have different precision/guarantees (Figure 3.1).

Definition 3 (Decidable Approximations)**.** *Given an undecidable set P, any decidable set*

$Q \subseteq \mathbb{N}$ is a decidable approximation of P. In addition, if $Q \subset P$, then Q is called a decidable under-approximation of P; if $P \subset Q$, then Q is called a decidable over-approximation of P.

Fact 3. *If P is an undecidable set, Q is a decidable approximation of P, then $P \triangle Q$ is infinite.*

Proof. If $P \triangle Q$ is finite, because Q is decidable, P is also decidable by Fact 1, which contradicts the assumption that P is undecidable. □

3.3 Witnessable Problems

This section formally defines witnessable problems and presents three main results in Section 3.3.1, Section 3.3.2, and Section 3.3.3, respectively. Section 3.3.4 demonstrates that we can iteratively compute infinitely many imprecision witnesses for each decidable approximation.

Definition 4. *We say an undecidable problem $P \in \mathcal{P}(\mathbb{N})$ is witnessable if and only if there exists a partial computable function w_P, such that for any decidable approximation $Q \subseteq \mathbb{N}$ and any natural number q such that $\phi_q = \chi_Q$, $w_P(q)$ is defined and $w_P(q) \in P \triangle Q$. The partial computable function w_P is called a witness function of P, and $w_P(q)$ is called an imprecision witness of Q.*

The definition of w_P is general. First, it only depends on the problem P and works on any "implementation" (index) q of any decidable approximation Q. Second, it does not require the witness function w_P to be total computable: it only requires that w_P is defined on all indices of characteristic functions of decidable sets. This definition gives us more flexibility to construct such witness functions. In this chapter, however, all constructed witness functions are total computable functions. Third, it only requires the existence of w_P. Whether there exists computable functions mapping P to w_P depends on how P is represented. In particular, many undecidability proofs [1, 21, 36] rely on many-one reductions

from the halting problem or its complement. If P is given together with such a many-one reduction, then we can directly construct w_P from the given reduction (Theorems Theorem 2, Theorem 3, and Theorem 4).

Another important observation is that we can decide whether the imprecision witness is a false positive or a false negative. Indeed, because Q is decidable, if $\chi_Q(w_P(q)) = 0$, then $w_P(q) \in P \setminus Q$; if $\chi_Q(w_P(q)) = 1$, then $w_P(q) \in Q \setminus P$. This observation enables iterative imprecision witness computation described in Section 3.3.4.

3.3.1 Diagonal Halting Problem is Witnessable

Our proof idea is inspired by reconsidering the classical undecidability proof [1, 2] of the diagonal halting problem K based on diagonalization. Specifically, replacing the hypothetical decider for K with an actual decider for any of K's decidable approximation Q yields an index in $K \triangle Q$.

Theorem 2 (Witnessability of Halt)**.** $K = \{i \mid \phi_i(i) \downarrow\}$ *is witnessable.*

Proof. It is well-known that K is undecidable [1, 2]. For any decidable approximation $Q \subseteq \mathbb{N}$ and a natural number q such that ϕ_q is the characteristic function of Q, we construct a 2-ary partial computable function f using the universal function (interpreter) for all 1-ary partial computable functions (which interprets q as ϕ_q):

$$f(q, x) = \begin{cases} \uparrow & \text{if} \quad \phi_q(x) = 1 \\ 0 & \text{if} \quad \phi_q(x) = 0. \end{cases}$$

Because f is a 2-ary partial computable function, there exists an index j such that $f(q, x) = \phi_j^2(q, x)$ for all $q, x \in \mathbb{N}$. By the S-m-n theorem, there exists a 2-ary total computable function ψ such that $\phi_j^2(q, x) = \phi_{\psi(j,q)}(x)$ for all $q, x \in \mathbb{N}$. Next, we claim that $\psi(j, q) \in K \triangle Q$ by case analysis. To demonstrate this, we show that $\psi(j, q) \notin K \cap Q$ and $\psi(j, q) \notin (K \cup Q)^c$ both by contradiction, and these two facts imply $\psi(j, q) \in K \triangle Q$.

1. If $\psi(j,q) \in K \cap Q$, then naturally $\psi(j,q) \in Q$, so we have

$$\begin{aligned} & \phi_q(\psi(j,q)) = 1 \\ \implies \quad & f(q,\psi(j,q)) \uparrow \\ \implies \quad & \phi_j^2(q,\psi(j,q)) \uparrow \\ \implies \quad & \phi_{\psi(j,q)}(\psi(j,q)) \uparrow . \end{aligned}$$

   However, because $\psi(j,q) \in K$, we have $\phi_{\psi(j,q)}(\psi(j,q)) \downarrow$, which is a contradiction.

2. If $\psi(j,q) \in (K \cup Q)^c$, in particular $\psi(j,q) \notin Q$, so we have

$$\begin{aligned} & \phi_q(\psi(j,q)) = 0 \\ \implies \quad & f(q,\psi(j,q)) = 0 \\ \implies \quad & \phi_j^2(q,\psi(j,q)) = 0 \\ \implies \quad & \phi_{\psi(j,q)}(\psi(j,q)) = 0. \end{aligned}$$

   But on the other hand, since $\psi(j,q) \notin K$, we have $\phi_{\psi(j,q)}(\psi(j,q)) \uparrow$, which is a contradiction.

Combining the above two facts, we conclude that $\psi(j,q) \in K \triangle Q$. The witness function can be set as $w_K(q) = \psi(j,q)$. In particular, this w_K is a total function. □

The above theorem shows K is witnessable by constructing a valid witness function w_K. However, there exists more than one witness function for K as discussed in Section 3.7.1. Moreover, we show in Sections Section 3.3.2 and Section 3.3.3 that the class of witnessable problems is closed under complements and many-one reductions. In that sense, K is the starting point for deriving witnessable problems, but this does not mean that K is the only such starting point.

3.3.2 Witnessability is Closed under Complements

The proof expresses the witness function for the complement problem P^c using the witness function for the original problem P.

Theorem 3 (Complement Closure)**.** *If an undecidable problem P is witnessable, then its complement P^c is also witnessable.*

Proof. Suppose P is witnessable, and then there exists a partial computable function w_P such that for any decidable set Q and any natural number q such that ϕ_q computes χ_Q, $w_P(q)$ is defined and $w_P(q) \in P \triangle Q$. Now consider P^c. For any decidable set R and any natural number r such that ϕ_r computes χ_R, consider the following 2-ary partial computable function:

$$g(r,x) = \begin{cases} 0 & \text{if} \quad \phi_r(x) = 1 \\ 1 & \text{if} \quad \phi_r(x) = 0. \end{cases}$$

Without loss of generality, assume that $g(r,x) = \phi_l^2(r,x)$. By the S-m-n theorem, there exists a 2-ary total computable function ψ such that $\phi_l^2(r,x) = \phi_{\psi(l,r)}(x)$ for all $r, x \in \mathbb{N}$. Clearly, $\phi_{\psi(l,r)}(x)$ is the characteristic function of R^c. Now consider $w_P(\psi(l,r))$: according to the definition of w_P, $w_P(\psi(l,r))$ is defined and $w_P(\psi(l,r)) \in P \triangle R^c = P^c \triangle R$. Thus, we can set $w_{P^c}(r) = w_P(\psi(l,r))$. □

3.3.3 Witnessability is Closed under Many-One Reductions

Assume that $P_1 \leq_m P_2$ and P_1 has a witness function; we show that P_2 also has a witness function. The proof composes a decidable approximation Q_2 of P_2 with the reduction from P_1 to P_2; this gives a decidable approximation Q_1 of P_1. Because we assume P_1 has a witness function, we can use that witness function to compute an imprecision witness for Q_1, and finally convert it back to an imprecision witness for Q_2.

The main technique used in our proof of Theorem 4 shares the same insight with [37]'s proof showing that if A is a creative set, $A \leq_m B$, and B is c.e., then B is creative. But

[37]'s proof only targets one set relation (by definition, the productive function only considers c.e. *subsets* of the creative set's complement), while our proof handles symmetric difference, which covers all possible relations between a witnessable problem and its decidable approximations.

Theorem 4 (Many-One Reduction Closure)**.** *If an undecidable problem P_1 is witnessable, and $P_1 \leq_m P_2$, then P_2 is also witnessable.*

Proof. Because P_1 is witnessable, there exists a witness function w_{P_1} for P_1. With the assumption $P_1 \leq_m P_2$, let f be a many-one reduction (a total computable function) from P_1 to P_2. According to the definition of many-one reductions (Definition Definition 2), $\forall x \in \mathbb{N}, x \in P_1 \Leftrightarrow f(x) \in P_2$. To prove that P_2 is witnessable, we show that there exists a witness function w_{P_2}, such that for any computable set Q_2 and any natural number q_2 such that $\phi_{q_2} = \chi_{Q_2}$, $w_{P_2}(q_2)$ is defined and $w_{P_2}(q_2) \in P_2 \triangle Q_2$. Consider the total computable function $g(x) = \phi_{q_2}(f(x))$. It is a total computable function with function values in $\{0, 1\}$, so it is a characteristic function of some decidable set Q_1. We can construct the following partial computable function:

$$h(i_1, i_2, x) = \phi_{i_1}(\phi_{i_2}(x)).$$

Suppose the index of h is j. By the S-m-n theorem, there exists a 3-ary total computable function ψ such that $h(i_1, i_2, x) = \phi_{\psi(j,i_1,i_2)}(x)$ for all $i_1, i_2, x \in \mathbb{N}$. Because f is known, suppose it has an index k, and we have $g(x) = h(q_2, k, x) = \phi_{\psi(j,q_2,k)}(x)$. Thus, we obtained an index $q_1 = \psi(j, q_2, k)$ for χ_{Q_1}. By the definition of w_{P_1}, $w_{P_1}(q_1) \in P_1 \triangle Q_1$. Now we claim that $f(w_{P_1}(q_1)) \in P_2 \triangle Q_2$.

1. If $f(w_{P_1}(q_1)) \in P_2 \cap Q_2$, then naturally we also have $f(w_{P_1}(q_1)) \in Q_2$, so $w_{P_1}(q_1) \in f^{-1}(Q_2) = Q_1$. But on the other hand, because $f(w_{P_1}(q_1)) \in P_2$, according to the definition of f, $w_{P_1}(q_1) \in P_1$. Combining those two, we have $w_{P_1}(q_1) \in P_1 \cap Q_1$, which is a contradiction because $w_{P_1}(q_1) \in P_1 \triangle Q_1$.

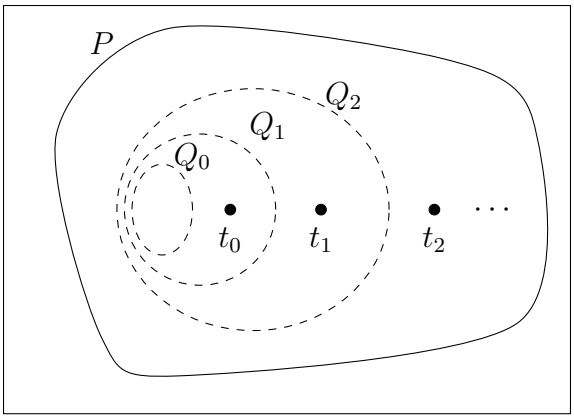


Figure 3.2: The iterative imprecision witness computation. For an undecidable problem P, starting from a decidable under-approximation Q_0, after computing an imprecision witness t_0 for Q_0, we incorporate t_0 to get a better approximation Q_1, and compute an imprecision witness t_1 for Q_1, and so on. The big rectangle surrounding P represents the set of all natural numbers. Other kinds of approximations (Figure 3.1) are similar.

2. If $f(w_{P_1}(q_1)) \in (P_2 \cup Q_2)^c$, then we have $w_{P_1}(q_1) \notin f^{-1}(Q_2) = Q_1$. According to the definition of f, we also have $w_{P_1}(q_1) \notin P_1$. Combining those two, we have $w_{P_1}(q_1) \in (P_1 \cup Q_1)^c$, which contradicts the fact that $w_{P_1}(q_1) \in P_1 \triangle Q_1$.

Clearly, $w_{P_2}(x) = f(w_{P_1}(\psi(j, x, k)))$ is a witness function for P_2, so P_2 is witnessable. □

There are many undecidable problems that can be proved by many-one reductions from K. In particular, all non-trivial semantic properties of programs mentioned in Rice's theorem are many-one reducible from K [1], and thus they are all witnessable according to the above theorem. Formally, an *index set* I is a subset of $\mathbb{N}$ satisfying $\forall i \in I, \forall j \in \mathbb{N}, (\phi_i = \phi_j \implies j \in I)$. An index set I is non-trivial if and only if $I \neq \emptyset$ and $I \neq \mathbb{N}$.

Corollary 1. *All non-trivial index sets are witnessable.*

Proof. Since there exists a many-one reduction from K to any non-trivial index set [1], this corollary immediately follows from Theorem 2 and Theorem 4. □

3.3.4 Iterative Imprecision Witness Computation

This section shows that we can compute infinitely many imprecision witnesses for each approximation of each witnessable problem: computing an imprecision witness, incor-

porating it into the approximation (in possibly naive ways), and repeating this process. Figure 3.2 illustrates this process based on under-approximations.

Note that a similar iterative process can also be done for productive sets and their c.e. subsets [16], but in that case, the iterative process can only be done for subsets by definition, while our construction uses symmetric difference to handle more general set relations. This requires our observation "$P \setminus Q$ and $Q \setminus P$ can be distinguished by the computable set Q," so that we can determine whether the witness is a false positive or a false negative.

Theorem 5 (Iterative Witnesses)**.** *If a problem P is witnessable and Q is a decidable approximation of P, then there is a 2-ary total computable function t such that for any ϕ_q computing χ_Q, $\{t(q,0), t(q,1), \ldots\}$ is an infinite list of different imprecision witnesses for Q.*

Proof. We describe how t works. First, $t(q,0)$ is defined as $w_P(q)$. Let $t_0 = t(q,0)$. We mainly discuss the case where $t_0 \in P \setminus Q$ and the other case ($t_0 \in Q \setminus P$) is similar. The two cases can be computably distinguished (because Q is decidable), so we can let t choose the correct case.

When $t_0 \in P \setminus Q$, we augment Q to obtain $Q' = Q \cup \{t_0\}$. The index for $\chi_{Q'}$ can be obtained using the following process. Consider the partial computable function g defined as follows:

$$g(y,z,x) = \begin{cases} 1 & \text{if } \ x = z \\ \phi_y(x) & \text{if } \ x \neq z. \end{cases}$$

Obviously, $h(x) = g(q, t_0, x)$ computes $\chi_{Q'}$. Suppose the index of g is j. By the S-m-n theorem we have a 3-ary total computable function ψ such that $g(y,z,x) = \phi_{\psi(j,y,z)}(x)$ for all $y, z, x \in \mathbb{N}$. Thus, $\psi(j,q,t_0)$ is an index for $\chi_{Q'}$. If we apply the witness function w_P for P again on $\psi(j,q,t_0)$, then we have $w_P(\psi(j,q,t_0)) \in P \triangle Q'$, so $t_1 = w_P(\psi(j,q,t_0)) \neq t_0$.

When $t_0 \in Q \setminus P$, the set $Q \setminus \{t_0\}$ is also computable, and a similar construction of t_1 can be done.

We define $t(q, 1)$ as t_1. For any $n \in \mathbb{N}$ and $n \geq 1$, repeating these computation steps n-times gives the definition of $t(q, n)$. □

The above theorem leads to an important implication: any witnessable problem must contain an infinite c.e. set because we can start from letting $Q = \emptyset$. This serves as a cornerstone for our construction of non-witnessable problems discussed in Section 3.4.

3.4 Non-Witnessable Problems

Section 3.3 shows that witnessable problems include many undecidable problems. In this section, we construct an undecidable problem that is non-witnessable.

Lemma 1. *Given a witnessable problem $P \in \mathcal{P}(\mathbb{N})$, there exists a computably enumerable set $E \subseteq P$, such that both E and E^c are infinite.*

Proof. Suppose P is witnessable and let $Q = \emptyset$ be a decidable under-approximation of P. According to Theorem 5, we can compute infinitely many imprecision witnesses $t_0, t_1, t_2, \cdots \in P \triangle Q = P$. This list is clearly computably enumerable, and we let $E = \{t_0, t_1, t_2, \cdots\} \subseteq P$. Because P^c is infinite by Fact 2 and $E^c \supseteq P^c$, it is immediate that E^c is infinite. □

Based on the above lemma, we can construct an undecidable problem X not containing any c.e. set E such that both E and E^c are infinite. These sets are known as *immune sets* [16]. Specifically, a set $I \subset N$ is immune if and only if I is infinite but does not contain any infinite c.e. set. Instead of directly adopting this definition, we provide our own construction to form a basis for proving Theorem 8 and make chapter more self-contained.

Our proof of Theorem 6 first lists all co-c.e. sets where both the sets and their complements are infinite (there are only countably many co-c.e. sets). Then, by a diagonalization-style construction, we can get a set X and prove that X is non-witnessable. The construction inherits some techniques of the construction in [38]. However, [38] constructs a simple set (*i.e.*, a c.e. set whose complement is immune), so its construction process needs

to be c.e. On the contrary, we construct an immune set directly and do not require the construction process to be c.e.

Theorem 6 (Non-Witnessable Problem)**.** *There exists a non-witnessable undecidable problem X.*

Proof. We list all co-c.e. sets $C_0, C_1, C_2, \cdots$ such that for all $i \in \mathbb{N}$, both C_i and C_i^c are infinite. It is easy to see that we can make this list $C_0, C_1, C_2, \cdots$ and also this list is infinite, because there are countably infinitely many c.e. sets such that both themselves and their complements are infinite. Now we construct a set Y by the following (infinite) process.

- Pick an arbitrary number y_0 from C_0^c, and include y_0 into Y.
- For each $i \in \mathbb{N}$ and $i \geq 1$, pick an arbitrary number y_i from C_i^c such that $y_i > y_{i-1}+1$, and include y_i into Y.

This process is infinite because C_i^c is infinite for every $i \in \mathbb{N}$. Now let $X = Y^c$, and we claim that X is non-witnessable.

1. It is easy to see that both X and Y are infinite, because $Y = \{y_0, y_1, y_2, \cdots\}$ is an infinite set and because we ensure that $y_i > y_{i-1} + 1$, the infinite set $\{y_0 + 1, y_1 + 1, y_2 + 1, \cdots\}$ is not included in Y.
2. We then show that X is undecidable. Indeed, if X is decidable, then $Y = X^c$ is also decidable, and because decidable sets are co-c.e., we have $Y = C_j$ for some $j \in \mathbb{N}$. However, due to the construction, Y is different from every C_i (because we pick at least one element from C_i^c and include it into Y), which is a contradiction.
3. Finally, we show that X does not contain any c.e. set E such that both E and E^c are infinite. Suppose there exists such an E, then $X \supseteq E$, $Y \subseteq E^c$, and E^c is an infinite co-c.e. set. Therefore, there exists a $j \in \mathbb{N}$ such that $Y \subseteq C_j$. However, due to the construction of Y, we know that Y contains at least one element from every C_i^c, which is a contradiction.

Lemma 1 claims that every witnessable problem must contain an infinite c.e. subset whose complement is also infinite, but X does not contain such a subset, so X cannot be witnessable. □

Explicitly constructing different kinds of "imprecision witnesses" is prevalent in mathematical logic. For example, Cantor's theorem states that for any set S, the cardinality of $\mathcal{P}(S)$ is strictly greater than the cardinality of S, and the typical proof is for any given injection f from S to $\mathcal{P}(S)$, we explicitly construct a set $\{x \in S \mid x \notin f(x)\}$ that is not in f's range [20]. As another example, in the classical proof of Gödel's first incompleteness theorem [22], for each formal system satisfying the conditions of that theorem, we can explicitly construct a "Gödel sentence" that is neither provable nor disprovable from the axioms of the formal system. In our case, witness functions only exist for witnessable problems.

3.5 Cardinalities of the Two Classes of Problems

The classes of witnessable problems and non-witnessable problems are both large. This section discusses the flexibility of constructing witnessable problems and non-witnessable problems and then proves that both classes of problems have cardinality $2^{\aleph_0}$.

We show that the class of witnessable problems has cardinality $2^{\aleph_0}$ based on the fact that this class is closed under many-one reductions (Theorem 4). Indeed, for the diagonal halting problem K, it is easy to construct a many-one reduction f from K to another problem such that $\mathbb{N} \setminus (f(K) \cup f(K^c))$ is infinite. The exact boundary between $f(K)$ and $f(K^c)$ can vary a lot: we can construct continuum many problems reducible from K. An immediate consequence is that many witness functions are shared by different witnessable problems, because the number of witness functions is countable.

Theorem 7 (Witnessable Class' Cardinality)**.** *There are $2^{\aleph_0}$ witnessable problems.*

Proof. By Theorem 2 and Theorem 4, we know that any problem P such that $K \leq_m P$

is witnessable. We show that there are continuum many such problems. First, it is easy to construct a decidable set H such that H is infinite and $H \subset K$. This could be done, for example, by letting H be all Gödel numbers of terms that do not use the unbounded minimization operator. Pick a Gödel number $j \in K \setminus H$ (so ϕ_j is a total recursive function and the term corresponding to j uses the unbounded minimization operator) and consider the following total computable function:

$$f(x) = \begin{cases} j & \text{if} \quad x \in H \\ x & \text{else.} \end{cases}$$

It is easy to verify that $f(K) = K \setminus H$ and $f(K^c) = K^c$. Now, we can map the elements in the set $2^{\mathbb{N}}$ (whose elements are countably infinite sequences of 0's and 1's) to the subsets of H in a straightforward way, using an injection η from $2^{\mathbb{N}}$ to $\mathcal{P}(H)$. For each point s in $2^{\mathbb{N}}$, we obtain a unique witnessable set $f(K) \cup \eta(s)$, because $K \leq_m f(K) \cup \eta(s)$ by the above total computable function f. On the other hand, it is clear that there is an injection from the class of witnessable problems to $\mathcal{P}(\mathbb{N})$. Because both $\mathcal{P}(H)$ and $\mathcal{P}(\mathbb{N})$ have the cardinality $2^{\aleph_0}$, due to the Cantor-Bernstein theorem [20], the cardinality of the class of witnessable problems is $2^{\aleph_0}$. □

The fact that there are continuum many non-witnessable problems is established by considering the diagonalization-style construction of X in Theorem 6: we are free to tweak the choice of each element in X so that we have two choices on each step. This gives continuum many versions of X.

Theorem 8 (Non-Witnessable Class' Cardinality)**.** *There are* $2^{\aleph_0}$ *non-witnessable problems.*

Proof. Similar to the proof of Theorem 7, we only need to construct an injection from $2^{\mathbb{N}}$ to the class of non-witnessable problems. To this end, we generalize the construction of the set Y in Theorem 6 as follows.

- Pick two arbitrary numbers $y_{0,0}$ and $y_{0,1}$ from C_0^c, and include either $y_{0,0}$ or $y_{0,1}$ into Y.

- For each $i \in \mathbb{N}$ and $i \geq 1$, pick two arbitrary numbers $y_{i,0}$ and $y_{i,1}$ from C_i^c such that $\min(y_{i,0}, y_{i,1}) > \max(y_{i-1,0}, y_{i-1,1}) + 1$, and include either $y_{i,0}$ or $y_{i,1}$ into Y.

Because there are two choices for each step and there are countably infinitely many steps for constructing Y, we can easily correspond different elements in $2^{\mathbb{N}}$ with different constructed versions of Y, resulting in different versions of $X = Y^c$. This is indeed an injection from $2^{\mathbb{N}}$ to the class of non-witnessable problems, and it completes the proof. □

Because the two classes of undecidable problems have the same cardinality, we can regard these two classes as "having the same size." Because witnessable problems can be regarded as having a computable property (we can computably construct imprecision witnesses for any given approximation), this fact contrasts with decidable sets: the cardinality of the class of decidable sets is $\aleph_0$, which is strictly "smaller" than the cardinality of the class of undecidable sets ($2^{\aleph_0}$).

3.6 Case Studies

This section presents two examples to demonstrate witness constructions using a simple programming language (defined in Section 3.6.1). Specifically, we convert the code of a sound sign analyzer to a program on which the analyzer is imprecise (Section 3.6.3), and convert the code of a sound string solver to a string formula on which the solver is imprecise (Section 3.6.4).

The constructions are not restricted to the two case studies. Moreover, the constructions are independent of the implementations of the program analyzers and SMT solvers. The analyzers and solvers do not need to be sound or complete. The only requirement is that they are written in Turing-complete programming languages and are total. In practice, both program analyzers and SMT solvers can be designed to run forever on certain cases, but

it is easy to convert them to total programs by reporting "unknown" when their executions exceed certain resource limits.

Finally, the constructions discussed in this section do not intend to be the most cost-effective realizations to be used in practice, but show the theoretical possibility of computing such imprecision witnesses. Also, Section 3.7.1 shows that we can apply code optimizations on many steps during the construction, which creates more possible ways to realize this construction.

3.6.1 The Lang Programming Language

We discuss the construction based on a simple, but Turing-complete dynamically-typed programming language Lang defined in Figure 3.3. Lang resembles a very small subset of Racket [39]. Lang supports two basic types: (unbounded) integers and strings. Data structures definable in other programming languages can be encoded to (or decoded from) integers or strings.

The term ($\mathcal{F}$ e$*$) in Lang's definition represents all basic operations on basic types (such as integer arithmetic and comparisons). In the extreme case, we can stipulate $\mathcal{F}$ to represent all total computable functions, which is similar to [40]'s language definition.

We show our construction using Lang, but our construction is completely language-agnostic, *i.e.*, specific language features such as syntax, semantics, and type systems do not affect our construction as long as the language is Turing-complete.

3.6.2 Overview of Constructions

Figure 3.4 gives an overview of our case studies. Without loss of generality, we assume the analyzers and solvers are sound, meaning that they give under-approximations for the corresponding decision problems. Similar constructions can always be done for other kinds of approximations. Our construction consists of five steps based on our proofs of Theorem 2, Theorem 3, and Theorem 4.

$$
\begin{array}{rcl}
\texttt{var} & \in & \text{Var} \\
\texttt{e} & ::= & \texttt{var} \mid a \in \text{Int} \mid s \in \text{Str} \mid (\ \textbf{lambda}\ (\ \texttt{var}*\ )\ \texttt{e}\ ) \\
& \mid & (\ \mathcal{F}\ \texttt{e}*\ ) \\
& \mid & (\ \textbf{letrec}\ (\ (\ \texttt{var}\ \texttt{e}\ )*\ )\ \texttt{e}\ ) \\
& \mid & (\ \textbf{if}\ \texttt{e}\ \texttt{e}\ \texttt{e}\ ) \\
& \mid & (\ \textbf{call}\ \texttt{e}\ \texttt{e}*\ )
\end{array}
$$

(a) Syntax of Lang. The notation "$*$" means the preceding symbol or parenthesized symbols occur zero or more times.

$$
\begin{array}{rcl}
[\![\text{env}, \texttt{var}]\!] & = & \text{deref}[\text{env}[\texttt{var}]] \\
[\![\text{env}, a]\!] & = & \text{a} \\
[\![\text{env}, s]\!] & = & \text{s} \\
[\![\text{env}, (\textbf{lambda}\ (\texttt{var}*)\ \texttt{e})]\!] & = & \langle \text{env}, \langle \texttt{var}*, \texttt{e} \rangle \rangle \\
[\![\text{env}, (\mathcal{F}\ \texttt{e}*)]\!] & = & \mathcal{F}[[\![\text{env}, \texttt{e}]\!]*] \\
[\![\text{env}, (\textbf{letrec}\ ((\texttt{var}\ \texttt{e}_1)*)\ \texttt{e}_2)]\!] & = & [\![\text{env} + \text{bindrec}[\text{env}, \texttt{var}*, \texttt{e}_1*], \texttt{e}_2]\!] \\
[\![\text{env}, (\textbf{if}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3)]\!] & = & \text{ite}[[\![\text{env}, \texttt{e}_1]\!], \text{env}, \texttt{e}_2, \texttt{e}_3] \\
[\![\text{env}, (\textbf{call}\ \texttt{e}_1\ \texttt{e}_2*)]\!] & = & [\![\text{env}_0 + \{(\texttt{var} \mapsto \text{new}[[\![\text{env}, \texttt{e}_2]\!]])*\}, \texttt{e}_0]\!] \\
& \text{where} & \langle \text{env}_0, \langle \texttt{var}*, \texttt{e}_0 \rangle \rangle = [\![\text{env}, \texttt{e}_1]\!] \\
 & & \\
\text{bindrec}[\text{env}, \texttt{var}*, \texttt{e}*] & = & (\text{deref}[\text{env}_1[\texttt{var}]] \leftarrow [\![\text{env} + \text{env}_1, \texttt{e}]\!])*; \text{env}_1 \\
& \text{where} & \text{env}_1 = \{(\texttt{var} \mapsto \text{new}[\text{undefined}])*\} \\
\text{ite}[\text{true}, \text{env}, \texttt{e}_1, \texttt{e}_2] & = & [\![\text{env}, \texttt{e}_1]\!] \\
\text{ite}[\text{false}, \text{env}, \texttt{e}_1, \texttt{e}_2] & = & [\![\text{env}, \texttt{e}_2]\!] \\
\text{new}[\text{val}] & = & \text{Memory} \leftarrow \text{Memory} \cup \{\text{loc} \mapsto \text{val}\}; \text{loc} \\
\text{deref}[\text{loc}] & = & \text{Memory}[\text{loc}]
\end{array}
$$

(b) Semantics of Lang. $[\![\text{env}, \texttt{e}]\!]$ means the evaluation result of the expression e in the environment env (mapping variables to memory locations). The order in env realizes variable shadowing. The initial environment is empty. $\langle\rangle$ means pairs. $a; b$ means sequencing (evaluating from left to right and returning the last value).

Figure 3.3: Syntax and semantics of the simple programming language Lang. Any case not defined in the semantics is considered invalid where the program is treated as divergent (non-terminating).

1. *Problem Construction (Figure 3.4a).* Construct a target decision problem D.

2. *Problem Reduction (Figure 3.4b).* Construct a many-one reduction from the diagonal halting problem K or its complement K^c to D.

3. *Approximation Construction (Figure 3.4c).* Propagate the under-approximation of D to an under-approximation of K or K^c.

4. *Witness Construction (Figure 3.4d).* Construct an imprecision witness in K or K^c.

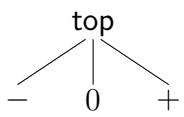
5. *Witness Mapping (Figure 3.4e).* Map the imprecision witness in K or K^c back to an imprecision witness in D.

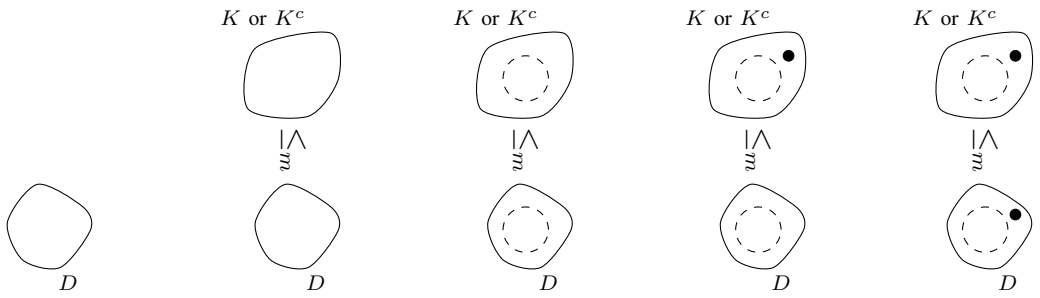
Theorem 2 give the starting point for constructing witnesses, and Theorem 3 and Theorem 4 gives the flexibility to propagate witness constructions along complements and many-one reductions. Our construction steps follow these theorems. Overall, the construction steps specify an algorithm (the witness function) taking as input the under-approximating program for D (which is obtained by simply wrapping the code of the given program analyzer/SMT solver), and producing an output on which the original program analyzer/SMT solver is imprecise. Once the problem D and the many-one reduction from K or K^c to D is fixed, this algorithm is also fixed, which is independent of any specific under-approximating program analyzers/SMT solvers.

3.6.3 Case Study 1: Program Analyzers

Given the code of a sound sign analyzer for Lang programs, we construct a program on which this analyzer is imprecise.

Analyzer Model. We stipulate that a sign analyzer takes as input a program and returns the following analysis results.





(a) Problem. (b) Reduction. (c) Approximation. (d) Witness. (e) Mapping back.

Figure 3.4: Construction overview: (a) constructing a decision problem D, (b) constructing a many-one reduction $\leq_m$ from K to D, (c) constructing an approximation for K (the dashed circle in K) based on the approximation for D (the dashed circle in D), (d) constructing a witness in K (the black point in K), (e) mapping the witness in K back to a witness in D (the black point in D).

Specifically, "$-$," "0," and "$+$" denote that the input program, on every input, always halts and produces a negative integer, the integer 0, and a positive integer, respectively. The special value "top" indicates either the program does not satisfy any of the aforementioned cases or the sign analyzer is unable to determine the program's output sign. In particular, practical program analyzers typically produce error messages on invalid programs, and our construction can wrap those analyzers so that they return "top" on invalid programs.

The Construction. Our construction requires a sound sign analyzer written in Lang for Lang programs and a Lang interpreter written in Lang. Their internal implementations are unconstrained. The sign analyzer is a (total) lambda taking an arbitrary Lang lambda (represented as a string) as input, and outputting "$+$," "$-$," "0," or "top."

```
1 (lambda (program)
2     (...code_of_analyzer...))
```

The Lang interpreter is a (partial) lambda that takes as input a single-parameter Lang lambda (represented as a string) with its input (also represented as a string), and returns the execution result of running the input lambda on the input (if it terminates). If the input is invalid[4] or a runtime error occurs during the interpretation, the interpreter enters an infinite loop.

[4]The interpreter can also perform static type checking before the execution.

```
(lambda (program, input)
    (...code_of_interpreter...))
```

Step 1: Problem Construction We convert the sign analysis problem to the problem D of verifying whether the input program returns a positive integer on every input. The sign analyzer can be converted to a program verifier for D, which is shown as follows. The return values of the verifier could be "correct" or "unknown." By the soundness of the sign analyzer, the verifier is an under-approximation of D. Note that "=" is one of the basic operations $\mathcal{F}$ in Lang's definition.

```
(lambda (program)
    (letrec ((analyzer
        (lambda (program)
            (...code_of_analyzer...))))
        (if (= (call analyzer program) "+")
            "correct"
            "unknown")))
```

Step 2: Problem Reduction We construct a many-one reduction from the diagonal halting problem to the problem D constructed in Step 1. The reduction, assuming that the input is the code of a program p_1, returns the code of another program p_2 such that p_2 returns a positive integer on every input if and only if p_1 terminates on itself. The function `format` denotes filling placeholders (`[]`) in a string with extra string arguments (preserving quotes). For example, `(format "A[]C" "b")` evaluates to `"A\"b\"C"`. Note that `format` is one of the basic operations $\mathcal{F}$ in Lang's definition.

```
(lambda (program)
    (format
        "(lambda (input)
            (letrec ((interpreter
                (lambda (program, input)
                    (...code_of_interpreter...))))
                (if (call interpreter [] [])
                    1
                    0)))"
```

```
10          program program))
```

Step 3: Approximation Construction Using the reduction in Step 2, we convert the verifier in Step 1 to the following program q that under-approximates K.

```
1 (lambda (program)
2     (letrec ((verifier (...code_of_verifier...))
3         (reduction (...code_of_reduction...)))
4         (if (= (call verifier (call reduction program)) "correct")
5             "terminating"
6             "unknown")))
```

Step 4: Witness Construction Now we have the code of q that under-approximates K. Based on the proof of Theorem 2, we construct the following imprecision witness (program) witnessing the imprecision of q: the code of this program is in K, but q returns "unknown" on it.

```
1 (lambda (program)
2     (letrec ((q (...code_of_q...)))
3         (if (= (call q program) "terminating")
4             (letrec ((loop (lambda () (call loop)))) (call loop))
5             0)))
```

Step 5: Witness Mapping According to the definition of the reduction in Step 2, the returned string of the following function call is an imprecision witness for the verifier that we constructed in Step 1, meaning that this witness terminates and returns a positive integer on every input, but the verifier returns "unknown" on it. As a result, the sign analyzer returns "top" on it.

```
1 (letrec ((reduction (...code_of_reduction...)))
2     (call reduction "(...code_of_the_witness_for_K...)"))
```

3.6.4 Case Study 2: SMT Solvers

This section discusses our construction for a specific type of SMT solvers: string solvers. Specifically, we consider the validity problem of the set of sentences written as a $\forall\exists$ quantifier alternation applied to positive word equations described in [36]'s work. For simplicity, we use $\mathbb{S}$ to denote the set of such sentences. Given the code of a sound solver for the validity problem of $\mathbb{S}$ sentences, we construct a valid $\mathbb{S}$ sentence on which the solver cannot conclude its validity.

Solver Model. We stipulate that a string solver takes an $\mathbb{S}$ sentence and returns either "valid," "invalid," or "unknown." Because the validity problem of $\mathbb{S}$ sentences is undecidable [36], any such solver must return "unknown" for some actually valid sentences.

The Construction. Our construction only requires a sound $\mathbb{S}$ solver written in Lang. The solver is a (total) lambda taking an $\mathbb{S}$ sentence (represented as a string) as input and outputting "valid," "invalid," or "unknown." If the input is not an $\mathbb{S}$ sentence, the solver should return "unknown."

```
(lambda (sentence)
    (...code_of_solver...))
```

Step 1: Problem Construction The first step is to construct a decision problem D: the validity problem of $\mathbb{S}$ sentences. Because we assume the solver can return three different values, we wrap it into a lambda that returns only two values: "valid" or "unknown." By the soundness of the original solver, the wrapped solver results in an under-approximation of D.

```
(lambda (sentence)
    (letrec ((solver
        (lambda (sentence)
            (...code_of_solver...))))
        (if (= (call solver sentence) "valid")
            "valid"
            "unknown")))
```

Step 2: Problem Reduction The second step is to construct a many-one reduction from K^c to the problem D constructed in Step 1. This step is based on a result due to [36]. Specifically, given a two-counter machine M and a finite string w, [36] construct an $\mathbb{S}$ sentence such that M does not halt on w if and only if this sentence is valid, and we denote this construction as f. For simplicity, we encode M and w into a single string Mw. On the other hand, because two-counter machines can simulate arbitrary Turing machines [36], we also have a computable function g such that for any pair of Lang program and input (L, v), both of which are represented as strings, $g((L, v)) = Mw$ is a string encoding a two-counter machine and its input such that M applied to w behaves the same as L applied to v. Finally, we construct the following reduction from K^c to D using f and g. Specifically, the input Lang program does not halt on itself if and only if the output is a valid $\mathbb{S}$ sentence.

```
(lambda (program)
    (letrec ((f (lambda (Mw) (...code_of_f...)))
        (g (lambda (L v) (...code_of_g...))))
        (call f (call g program program))))
```

Step 3: Approximation Construction Using the reduction in Step 2, we convert the wrapped solver in Step 1 to the following program q that under-approximates K^c.

```
(lambda (program)
    (letrec ((wrapped_solver (...code_of_wrapped_solver...))
        (reduction (...code_of_reduction...)))
        (if (= (call wrapped_solver (call reduction program)) "valid")
            "non-terminating"
            "unknown")))
```

Step 4: Witness Construction Now we have the code of q that under-approximates K^c. Based on the proofs of Theorem 2 and Theorem 3, we construct the following imprecision witness (program) witnessing the imprecision of q: its code is in K^c but q returns "unknown" for it.

```
(lambda (program)
    (letrec ((q (...code_of_q...)))
```

```
3         (if (= (call q program) "non-terminating")
4             0
5             (letrec ((loop (lambda () (call loop)))) (call loop)))))
```

Step 5: Witness Mapping According to the definition of the many-one reduction in Step 2, the returned string of the following function call is an imprecision witness for the wrapped solver that we constructed in Step 1, meaning that it is a valid $\mathbb{S}$ sentence but the wrapped solver returns "unknown" on it. As a result, the original solver also returns "unknown" on it.

```
1 (letrec ((reduction (...code_of_reduction...)))
2     (call reduction "(...code_of_the_witness_for_Kc...)"))
```

3.7 Discussions

3.7.1 The Flexibility of Constructing Imprecision Witnesses

In general, the construction of imprecision witnesses is flexible. For example, for a specific undecidable problem P such that $K \leq_m P$ (via the many-one reduction f), the construction of imprecision witnesses for a decidable approximation Q of P is parameterized by at least the following factors:

- Different programs (indices) of Q;
- Different programs (indices) of $f^{-1}(Q)$; and
- Different reductions f from K to P.

In particular, we can apply program optimizations on the program of Q and the program of $f^{-1}(Q)$. In addition, Theorem 5 states that we can perform iterative construction of imprecision witnesses indefinitely, where the method to "improve" the approximation at each step is also flexible.

Moreover, as mentioned in Section 3.3.1, K might not be the only starting point of the reduction. In particular, the construction of imprecision witnesses for K might be

generalized to other diagonalization-based undecidability proofs, which we leave for future work.

3.7.2 The Classification of Undecidable Problems

Degree structures (*e.g.*, many-one degrees, and Turing degrees [1]) are commonly used to classify problems based on their relative computability. Our witnessable/non-witnessable problems, on the other hand, classify undecidable problems based on their "computable approximability": witnessable problems admit computable approximation improvements, while non-witnessable problems do not. We can still design decidable approximations for non-witnessable problems, but we do not have computable ways to automatically find imprecision witnesses.

We also compare our definition with several other classes of sets in computability theory. First, the class of witnessable problems is indeed different from the class of productive sets [16] (despite the similarities between their definitions): productive sets cannot be recursively enumerable, but the recursively enumerable set K is witnessable. Second, it is easy to see that our class of witnessable problems is not contained in the class of all c.e. sets, because we prove that the cardinality of the class of witnessable problems is $2^{\aleph_0}$ while there are only countably many c.e. sets. One direct implication is that, in general, the infinite sequence of imprecision witnesses constructed in Theorem 5 may not cover all points in the undecidable problem being approximated (otherwise, the problem is c.e.). Third, our witnessable problems are different from the concept of limit computability. Indeed, the limit lemma [41] states that a problem P is limit computable if and only if $P \leq_T \emptyset'$, *i.e.*, P is Turing reducible to the first Turing jump of the empty set, where actually $\emptyset' = K$. Our class of witnessable problems includes the set of indices of all total functions $L = \{i \mid \forall x \in \mathbb{N}, \phi_i(x) \downarrow\}$ (because L is a non-trivial index set and thus is many-one reducible from K), and the Turing degree of L is $\mathbf{0}''$, so L is not Turing-reducible to $\emptyset'$.

3.7.3 Non-Witnessable Problems in Practice

Section 3.4 explicitly constructs a non-witnessable problem but does not relate that problem to any real scenarios in programming language theory and related fields. The spirit of that construction is similar to constructions in typical proofs of the time/space hierarchy theorems [2]: the constructed problems' main goal is to serve as a theoretical example to support the theorem instead of modeling any practical scenarios.

3.7.4 A Counter-Intuitive Fact: "Harder" Problems Do Not Prevent Witnessability

A counter-intuitive fact is that although many-one reduction is considered as a hardness comparison (*i.e.*, if $A \leq_m B$ then B is considered "harder" than A), accumulation of many-one reductions cannot make a witnessable problem hard enough to be non-witnessable. Imagine a finite but arbitrarily long chain of problems $P_0 <_m P_1 <_m P_2 <_m \cdots <_m P_n$, where $A <_m B$ is the strict version of $A \leq_m B$. Once we prove P_0 is witnessable, we know that P_n is still witnessable, despite that the many-one reductions show that P_n is much "harder" than P_0.

3.8 Related Work

Extensive work exists on the undecidability of problems in programming language theory and related fields [3, 4, 5, 6, 31, 32, 26, 27, 11]. Our work goes beyond that: we analyze the "computable approximability" of different problems and provides computable imprecision witnesses for decidable approximations of certain undecidable problems.

There also exists work focusing on intensional aspects of computability results [21, 42, 43]. Our result does not focus on extensional aspects or intensional aspects in particular, but rather on transforming the proofs of undecidability to witness functions. In other words, our result is applicable to both the traditional Rice's theorem [1] and some intensional versions of Rice's theorem [21].

[44] and [40] propose constructions of incomplete cases for abstract interpretation, and abstract interpretation has been shown to be quite general to cover some other apparently different techniques [45]. Our approach is even more general: we do not make any assumptions about what framework the program analyzer is based on (it could be based on abstract interpretation, but could also be based on arbitrary combinations of program analysis techniques [9, 46, 47] and arbitrary heuristics), and we do not require the program analyzer to be sound or complete.

In computability theory, the classes of problems that are similar to our class of witnessable problems include c.e. sets and limit computable sets, because they all describe certain kinds of "approximating" processes. In Section 3.7.2, we discussed the difference between those two classes and our class, showing that our witnessable problems are indeed a new class of problems. Our classification motivation is also different from classifications based on relative computability with respect to oracles (such as Turing degrees and m-degrees): we classify undecidable problems based on decidable approximability.

Some of our proofs share similar ideas and methods with existing work. First, diagonalization and many-one reductions are standard techniques in computability [1], but we apply them to the scenario of our new concept (witnessability). Our proofs of Theorem 4, Theorem 5, and Theorem 6 share similar ideas with existing work in creative sets [37] and simple sets [38]. However, our work targets the new concept (witnessability) and more general set relations (modeled by the symmetric difference). The intent of our paper is not simply an extension of the existing work. Instead, our focus is witnessability's implications in programming language theory and formal methods, which shows that real (undecidable) problems and their approximations have the previously unknown "witness producing" computability property.

The word "approximation" is also used in algorithm design: for optimization problems, we can design algorithms whose outputs approximate the optimal solution [48], and relevant approximability results are also developed [49]. In contrast, our work focuses on

decision problems instead of optimization problems, and we use decidable decision problems to approximate undecidable decision problems.

3.9 Chapter Conclusion

This chapter defined witnessable problems, which are undecidable problems having computable imprecision witnesses for arbitrary decidable approximations. The class of witnessable problems has the same cardinality as the class of all undecidable problems. In particular, almost all problems in programming language theory and formal methods are witnessable, and algorithms in those areas are essentially decidable approximations of witnessable problems. Our results justified the research efforts on decidable approximations of witnessable problems and show the existence of universal ways to improve such approximations.

CHAPTER 4

ON WITNESS FUNCTIONS FOR COMPLEXITY LOWER BOUNDS

4.1 Introduction

Consider separating a computational problem X from a complexity class $\mathbf{C}$. Let $\mathbf{C}$ be represented as a set of programs under the corresponding resource restrictions. From now on, we will use "problems in $\mathbf{C}$" and "programs in $\mathbf{C}$" interchangeably. The proposition $X \notin \mathbf{C}$ can be formalized as follows.

$$\forall p \in \mathbf{C}, \exists x, p(x) \neq X(x).$$

The input x, which witnesses the difference between any candidate program $p \in \mathbf{C}$ and X, typically depends on p. Thus, the proposition essentially claims the existence of a function $p \mapsto x$, which is called a *witness function*. If X is computable and $\mathbf{C}$ contains only total programs, then the proposition $X \notin \mathbf{C}$ itself implies the existence of a computable witness function: for any $p \in \mathbf{C}$, enumerate inputs and compare $p(x)$ and $X(x)$ until the first difference point is found. The existence of such computable witness functions is implicit in Kozen's work [14].

In this chapter, we consider additional properties of such witness functions for complexity class separations. We utilize Blum-like axiomatic approach [50, 51] for computational complexity, which makes our results largely machine-independent and complexity-class independent. A crucial observation is that most common complexity classes allow arbitrarily large constant additive/multiplicative factors, so we make the distinction between two types of representations of $\mathbf{C}$: the unlabeled representation where only the programs are given, and the labeled representation where every program is equipped with the constant denoting the specific additive/multiplicative factor for that program's resource upper bound. The we

prove two theorems relating witness functions and *universal reductions*, where a universal reduction relates a complexity class $\mathbf{C}$ and a computational problem X by reducing every problem $Y \in \mathbf{C}$ to X. The two theorems can be roughly stated as follows.

- For unlabeled representations, under mild conditions, if w is a witness function for any $X \notin \mathbf{C}$, then there exists a linear complexity function w such that $w \circ \mathsf{w}$ is a universal reduction from any other class $\mathbf{D}$ to X.

- For labeled representations, under mild conditions, if w is a witness function for any $X \notin \mathbf{C}$, then there exists a $O(n(\log n)^2)$ complexity function w such that $w \circ \mathsf{w}$ is a universal reduction from a slightly smaller class $\mathbf{C}'$ to X.

Kozen's work [14] contains a preliminary version of these theorems, but didn't analyze the detailed complexity nor distinguish the difference between unlabeled representations and labeled representations. As we will show in Section 4.6, the witness functions implicitly contained in well-known complexity theoretic proofs are labeled, i.e. they assume the knowledge of the exact constant additive/multiplicative factors.

We identify three implications from our results.

1. For unlabeled representations of complexity classes, there is no reasonable complexity upper-bound for any witness function w for $X \notin \mathbf{C}$. This means constant additive/multiplicative factors are important in constructive complexity class separation proofs.

2. For labeled representations of complexity classes, suppose w is a witness functions for $X \notin \mathbf{C}$. There is a fast universal reduction from any problem in $\mathbf{C}'$ to $X \circ w$. This alludes that although X itself may not be a problem directly encoding all programs in $\mathbf{C}'$, $X \circ w$ is directly encoding all programs in $\mathbf{C}'$, making $X \circ w$ be similar to the artificial problems constructed in time/space hierarchy theorems [17, 18]. In other words, any separation proof, which claims the existence of w, is implicitly showing the conversion of X to an artificial problem $X \circ w$.

3. We show that the proofs for the time/space hierarchy theorems [17, 18] on Turing machines implicitly contain labeled witness functions, which shows witness functions are real concepts existing in well-known complexity theoretic proofs.

4.2 Preliminary

4.2.1 Partial Functions from $\mathbb{N}^{<\omega}$ to $\mathbb{N}$

Let $\mathbb{N}^{<\omega} = \{(x_1, x_2, \ldots, x_n) \mid n \in \mathbb{N} \wedge x_i \in \mathbb{N}\}$. As usual, $\rightharpoonup$ denotes partial functions and $\rightarrow$ denotes total functions. Let $\alpha(f)$ denote the arity of the function f. We write $f = g$ for

$$\alpha(f) = \alpha(g) \wedge (\forall \vec{x}, (f(\vec{x}) \uparrow \wedge g(\vec{x}) \uparrow) \vee (f(\vec{x}) \downarrow \wedge g(\vec{x}) \downarrow \wedge f(\vec{x}) = g(\vec{x}))).$$

We assume natural numbers are represented in their binary forms, and use $\|\vec{x}\|$ to denote the total length of the binary representations of each element $\sum \|x_i\|$. Ordinary Turing machines' inputs and outputs encoded as binary strings can be stipulated to always start with a "1" and thus can be represented as natural numbers.

We use lightface symbols $\mathrm{A}, \mathrm{B}, \ldots$ to represent sets of natural numbers, and use boldface symbols $\mathbf{A}, \mathbf{B}, \ldots$ to represent sets of such sets.

4.2.2 The Programming System

Let the set of all partial computable functions $\mathbf{PF} = \{\phi_i : \mathbb{N}^{<\omega} \rightharpoonup \mathbb{N} \mid i \in \mathbb{N}\}$ be enumerated by an admissible numbering [52], where the indices i can be seen as programs in a fixed Turing-complete programming language. When the arity n of ϕ_i is clear from the context, we use $\forall i$ as an abbreviation for $\forall i \in \{j \in \mathbb{N} \mid \alpha(\phi_j) = n\}$. Furthermore, ϕ_i satisfies the following two well-known conditions [1].

- (Universal Functions) For any $n \in \mathbb{N}$, there exists a program $u_n \in \mathbb{N}$ such that

$$\forall i, \lambda \vec{x}.\phi_{u_n}(i, \vec{x}) = \lambda \vec{x}.\phi_i(\vec{x})$$

  where $\vec{x} \in \mathbb{N}^n$. We use u to denote u_1.

- (S-m-n) For any $m, n \in \mathbb{N}$, there exists a total computable function s_m^n such that

$$\forall i, \lambda \vec{x}\vec{y}.\phi_{s_m^n(i,\vec{x})}(\vec{y}) = \lambda \vec{x}\vec{y}.\phi_i(\vec{x}, \vec{y})$$

  where $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

Let $\mathbf{TF} \subsetneq \mathbf{PF}$ be the set of total computable functions. It is well-known [15] that there doesn't exist a computable enumeration e such that $\mathbf{TF} = \{\phi_{e(i)} \mid i \in \mathbb{N}\}$. Let $\mathbf{TF_1}$ be the set of 1-ary, $\{0, 1\}$-valued total computable functions, corresponding to the set of computable subsets of $\mathbb{N}$.

4.2.3 Abstract Complexity Measure

We utilize Blum's axiomatic complexity measure [50] for the complexity of programs. Concrete complexity measures such as time / space complexities for Turing machines satisfy the basic axioms of Blum's approach [50].

Definition 5. *A function $(\lambda i.\Phi_i) : \mathbb{N} \to \mathbf{PF}$ is a Blum (computational) complexity measure if and only if it satisfies the following two basic axioms.*

- $\forall i, \alpha(\phi_i) = \alpha(\Phi_i) \wedge (\forall \vec{x}, \phi_i(\vec{x}) \downarrow \Leftrightarrow \Phi_i(\vec{x}) \downarrow)$.
- *The set $\{\langle i, \vec{x}, t\rangle \mid \Phi_i(\vec{x}) = t\}$ is decidable.*

$\Phi_i(\vec{x})$ is the cost (time, space, etc.) of program i on input x. By definition, this cost is

a natural number and is thus non-negative. The big-O notation is defined as follows. [1]

$$f = O(g) \quad \Leftrightarrow \quad \exists C, \forall \vec{x}, g(\vec{x}) \downarrow \Rightarrow f(\vec{x}) \leq Cg(\vec{x}) + C.$$

Similar to the spirit of Asperti [51], axiomatization is not our focus, and we introduce convenient axioms that make sense on common cost models along our discussions. In particular, the following axioms are always assumed.

Axiom 1 (Redundancy)**.** *For any $m, n \in \mathbb{N}$, there exists a total computable function r_m^n satisfying*

$$\exists r, \phi_r = r_m^n \wedge \Phi_r = O(\lambda i.\|i\|);$$

$$\forall i, \begin{cases} \lambda \vec{x}\vec{y}.\phi_{r_m^n(i)}(\vec{x}, \vec{y}) & = & \lambda \vec{x}\vec{y}.\phi_i(\vec{x}) \\ \lambda \vec{x}\vec{y}.\Phi_{r_m^n(i)}(\vec{x}, \vec{y}) & = & O(\lambda \vec{x}\vec{y}.\Phi_i(\vec{x})) \end{cases}$$

where $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

This is saying that $r_m^n(i)$ can ignore the redundant $\vec{y}$ argument and behave similarly to i. In particular, when $n = 0$, choosing $r_m^0(i) = i$ satisfies the requirements. Axiom 1 also implies that constant functions can be implemented using constant complexity programs:

$$\begin{array}{ll} & \alpha(i) = 0 \wedge \phi_i() = C \\ \Rightarrow & \lambda \vec{y}.\phi_{r_0^n(i)}(\vec{y}) = \lambda \vec{y}.\phi_i() = C \wedge \lambda \vec{y}.\Phi_{r_0^n(i)}(\vec{y}) = O(\lambda \vec{y}.\Phi_i()) = \mathsf{constant}. \end{array}$$

Axiom 2 (Efficient S-m-n)**.** *For any $m, n \in \mathbb{N}$, there exists a total computable function s_m^n satisfying*

$$\exists s, \phi_s = s_m^n \wedge \Phi_s = O(\lambda i \vec{x}.\|i\| + \|\vec{x}\|);$$

$$\forall i, \begin{cases} \lambda \vec{x}\vec{y}.\phi_{s_m^n(i,\vec{x})}(\vec{y}) & = & \lambda \vec{x}\vec{y}.\phi_i(\vec{x}, \vec{y}) \\ \lambda \vec{x}\vec{y}.\Phi_{s_m^n(i,\vec{x})}(\vec{y}) & = & O(\lambda \vec{x}\vec{y}.\|\vec{x}\| + \Phi_i(\vec{x}, \vec{y})) \end{cases}$$

[1] There are other definitions for the multi-variable big-O notation, and we choose one that is suitable for our purposes.

where $\vec{x} \in \mathbb{N}^m$ *and* $\vec{y} \in \mathbb{N}^n$.

In real computation models such as Turing machines, the first condition corresponds to the cost of concatenating the source code i with the fixed input $\vec{x}$, and the second condition corresponds to the cost of copying the hardcoded arguments $\vec{x}$ onto the tape before invoking ϕ_i.

Axiom 3 (Efficient Composition)**.** *For any* $m, n \in \mathbb{N}$*, there exists a total computable function* h_m^n *satisfying*

$$\exists h, \phi_h = h_m^n \wedge \Phi_h = O(\lambda i \vec{j}.\|i\| + \|\vec{j}\|);$$

$$\forall i, \vec{j}, \begin{cases} \lambda\vec{x}.\phi_{h_m^n(i,\vec{j})}(\vec{x}) &= \lambda\vec{x}.\phi_i(\phi_{j_1}(\vec{x}), \ldots, \phi_{j_m}(\vec{x})) \\ \lambda\vec{x}.\Phi_{h_m^n(i,\vec{j})}(\vec{x}) &= O(\lambda\vec{x}.\Phi_i(\phi_{j_1}(\vec{x}), \ldots, \phi_{j_m}(\vec{x})) + \Phi_{j_1}(\vec{x}) + \ldots + \Phi_{j_m}(\vec{x})) \end{cases}$$

where $\vec{j} \in \mathbb{N}^m$ *and* $\vec{x} \in \mathbb{N}^n$.

In real computation models such as Turing machines, the first condition corresponds to the cost of concatenating source code fragments, and the second condition corresponds to the cost of applying ϕ_i on the outputs of $\phi_{j_1}, \ldots, \phi_{j_m}$. For space complexity of Turing machines, the bound of $\lambda\vec{x}.\Phi_{h_m^n(i,\vec{j})}(\vec{x})$ may be optimized to $O(\lambda\vec{x}.\max\{\Phi_i(\phi_{j_1}(\vec{x}), \ldots, \phi_{j_m}(\vec{x})), \Phi_{j_1}(\vec{x}) + \ldots + \Phi_{j_m}(\vec{x})\})$. But the above bound is sufficient for our purposes.

Axiom 4 (Efficient Branching)**.** *For any* $m \in \mathbb{N}$*, there exists a total computable function* b_m *satisfying*

$$\exists b, \phi_b = b_m \wedge \Phi_b = O(\lambda i j_1 j_2.\|i\| + \|j_1\| + \|j_2\|);$$

$$\forall i, j, \begin{cases} \lambda\vec{x}.\phi_{b_m(i,j_1,j_2)}(\vec{x}) &= \lambda\vec{x}.\begin{cases} \phi_{j_1}(\vec{x}) & \text{if } \phi_i(\vec{x}) \\ \phi_{j_2}(\vec{x}) & \text{else} \end{cases} \\ \lambda\vec{x}.\Phi_{b_m(i,j_1,j_2)}(\vec{x}) &= O\left(\lambda\vec{x}.\|x\| + \begin{cases} \Phi_{j_1}(\vec{x}) + \Phi_i(\vec{x}) & \text{if } \phi_i(\vec{x}) \\ \Phi_{j_2}(\vec{x}) + \Phi_i(\vec{x}) & \text{else} \end{cases}\right) \end{cases}$$

where $x \in \mathbb{N}^m$.

The branch condition is treated as false if and only if $\phi_i(\vec{x}) = 0$. The cost associated with the branching program $b_m(i, j_1, j_2)$ is also conditioned on which branch is taken. In both cases, the costs of duplicating the input and evaluating the branch condition are incorporated.

Axiom 5 (Interpretation Overhead)**.** *For any $n \in \mathbb{N}$, the complexity overhead introduced by the universal function ϕ_{u_n} can be expressed as*

$$\forall i \in \mathbb{N}, \forall \vec{x} \in \mathbb{N}^n, \Phi_{u_n}(i, \vec{x}) \leq \Gamma_n(\|i\| + \|\vec{x}\|) + \gamma_n(i) J_n(\Phi_i(\vec{x}))$$

where Γ_n is a constant, γ_n is a total function and J_n is a total, non-decreasing function. We use Γ, γ, J to denote Γ_1, γ_1, J_1.

The factor $\gamma(i)$ in the overhead is a fixed constant for every program i. For single tape Turing machines simulating other single tape Turing machines, it can be achieved that $J(y) = y$ by organizing two tracks of the tape [18], where one track holds the description and state of the machine being simulated (which is moved together with the head), and the other track actually holds the tape of the machine being simulated. Also note that when $\phi_i(\vec{x})$ is undefined, according to Blum's basic axioms (Definition 5), $\Phi_i(\vec{x})$ is also undefined.

Axiom 6 (Efficient Linear Operations)**.** *For any $m \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^m$, any finite linear expression $e(\vec{x})$ generated from the following grammar*

$$e := x_i \mid x_i + x_j \mid x_i - x_j \mid x_i < x_j \mid x_i \leq x_j \mid x_i > x_j \mid x_i \geq x_j \mid x_i = x_j \mid x_i \neq x_j$$

can be implemented in linear time:

$$\exists i, \phi_i = e \wedge \Phi_i = O(\lambda \vec{x}. \|\vec{x}\|).$$

Since natural numbers are assumed to be represented in binary forms, the cost is pro-

portional to $\|\vec{x}\| = O(\log x)$ if x is interpreted as a natural number.

4.3 Complexity Classes and Their Representations

4.3.1 Complexity Classes

We focus on the complexity of 1-ary, $\{0,1\}$-valued total computable functions ($\mathbf{TF_1}$), which correspond to computable decision problems. A direct way to define a complexity class of such decision problems is to pose an exact complexity bound $t : \mathbb{N} \to \mathbb{N}$:

$$\{f \in \mathbf{TF_1} \mid \exists i \in \mathbb{N}, \phi_i = f \wedge \forall x \in \mathbb{N}, \Phi_i(x) \leq t(x)\}.$$

However, since it is straightforward to hardcode finitely many function values in a program, the complexity on finitely many points of a function is not very interesting. Also, Turing machines' linear speedup theorem [53] shows that on such computation models, constant multiplicative factors are not essential. In our setting, we consider a generalized version of the well-known concept of asymptotic complexity [54]. Here the additional parameter k serves as additive/multiplicative factors or similar purposes, and we will see throughout this chapter that there are great flexibilities for defining the complexity function $t(k, x)$ with respect to k, even for the same complexity class.

Definition 6. *Given a* 2*-ary total computable function* $t \in \mathbf{TF}$*, define the corresponding complexity class as*

$$\mathbf{CC}(t) = \{f \in \mathbf{TF_1} \mid \exists i \in \mathbb{N}, \phi_i = f \wedge \exists k \in \mathbb{N}, \forall x \in \mathbb{N}, \Phi_i(x) \leq t(k, x)\}.$$

Example 3. *Fix the abstract complexity measure* $\lambda i.\Phi_i$ *to be the usual time complexity on*

deterministic Turing machines. We can define the usual complexity classes as follows.

$$\begin{aligned}
\forall d \in \mathbb{N}, \mathbf{TIME}(n^d) &= \mathbf{CC}(\lambda kx.k + k\|x\|^d) \\
\forall d \in \mathbb{N}, \mathbf{TIME}(d^n) &= \mathbf{CC}(\lambda kx.k + k \cdot d^{\|x\|}) \\
\mathbf{P} &= \mathbf{CC}(\lambda kx.k + \|x\|^k) \\
\mathbf{EXP} &= \mathbf{CC}(\lambda kx.2^{k+\|x\|^k})
\end{aligned}$$

Note that these usual complexity classes are defined only in terms of the input length, while Definition 5 is more general and can depend on specific input values of the same length. Under Definition 5, the usual linear complexity of a program i is expressed as $\Phi_i(x) = O(\|x\|) = O(\log x)$.

4.3.2 Representations of Complexity Classes

Complexity classes are sets of functions, but we often need concrete representations of those (usually infinite) sets. Given a complexity class, the most straightforward way is to consider the set of programs with the corresponding complexity upper bound. The same functions can also be computed by time-wasting programs exceeding the upper bound (e.g. manually adding redundant loops), but ignoring those programs will not change the complexity class being defined.

Because of the asymptotic behavior of complexity upper bounds, the complexity bound includes a constant k as defined in Definition 6. This constant often serves as the constant additive or multiplicative factors as shown in Example 3. Due to this phenomenon, we distinguish unlabeled representations and labeled representations of complexity classes, where the difference is whether the constant k is encoded in the representation.

Definition 7. *The unlabeled representation of a complexity class* $\mathbf{CC}(t)$ *is the following set.*

$$\mathrm{UR}(t) = \{i \in \mathbb{N} \mid \phi_i \in \mathbf{TF_1} \wedge \exists k \in \mathbb{N}, \forall x \in \mathbb{N}, \Phi_i(x) \leq t(k, x)\}$$

Definition 8. *The labeled representation of a complexity class* $\mathbf{CC}(t)$ *is the following set.*

$$\mathrm{LR}(t) = \{\langle i, k\rangle \in \mathbb{N}^2 \mid \phi_i \in \mathbf{TF_1} \wedge \forall x \in \mathbb{N}, \Phi_i(x) \leq t(k, x)\}$$

$\mathrm{UR}(t)$ and $\mathrm{LR}(t)$ are not required to be computable. We can also consider computable sets of complexity-bounded programs, such as the set of Turing machines associated with step counters and polynomial step upper bounds, where the step counter terminates the program with a default return value when the number of steps exceeds the limit. But for our purposes, the above definitions are sufficient.

4.4 Witness Functions and Universal Reductions

In this section, we define the concept of witness functions, whose existence is equivalent to complexity lower bounds. We then define an auxiliary concept called universal reductions, which generalizes many-one reductions.

4.4.1 Witness Functions

Complexity lower bounds correspond to non-memberships of total computable functions with respect to complexity classes. For example, the proposition $\mathbf{P} \neq \mathbf{NP}$ is equivalent to $\mathrm{SAT} \notin \mathbf{CC}(\lambda kx.k + \|x\|^k)$. The non-membership can be equivalently expressed as having a witness function, which is implicit in Kozen's work [14].

Definition 9. *Given a function* $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$*, a partial function* $w : \mathbb{N} \rightharpoonup \mathbb{N}$ *is an unlabeled witness function for the non-membership* $g \notin \mathbf{CC}(t)$ *if and only if*

$$\forall i \in \mathrm{UR}(t), \phi_i(w(i)) = \neg g(w(i));$$

a partial function $w : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ *is a labeled witness function for the non-membership*

$g \notin \mathbf{CC}(t)$ *if and only if*

$$\forall \langle i, k \rangle \in \mathrm{LR}(t), \phi_i(w(i,k)) = \neg g(w(i,k)).$$

The negation operator $\neg$ naturally means flipping $0/1$. The witness function w gives, for every program i in the representation of the complexity class, an input on which ϕ_i and g have different values. There could be many different witness functions for the same non-membership relation $g \notin \mathbf{CC}(t)$. In particular, assuming $g \notin \mathbf{CC}(t)$, there exists the following naive, unlabeled witness function w that is computable on the domain $\mathrm{UR}(t)$.

Procedure 1. *Suppose* $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$. *For* $i \in \mathrm{UR}(t)$, *compute* $w(i)$ *as follows: enumerate all inputs* x *and compare* $\phi_i(x)$ *and* $g(x)$ *until we find the first different point. Since both of* ϕ_i *and* g *are total, and* $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$, *this procedure will terminate on every* $i \in \mathrm{UR}(t)$.

The complexity of this naive procedure, however, could be high if $\mathbf{CC}(t)$ "approximates" g well.

Labeled witness functions are used in classical complexity theoretic proofs, such as the typical proof of time / space hierarchy theorems [17, 18]. In other words, those proofs need the knowledge of the specific k in the complexity bounds in order to construct the desired witness. We will prove a generalized hierarchy theorem and analyze its witness function in Section 4.6.

4.4.2 Universal Reductions

On the other hand, various types of reductions are the common way to compare the hardness of problems. Specifically, for two decision problems $\mathrm{A}, \mathrm{B} \subseteq \mathbb{N}$, a many-one reduction $r \in \mathbf{TF}$ from A to B satisfies $\forall x \in \mathbb{N}, \mathrm{A}(x) \Leftrightarrow \mathrm{B}(r(x))$. We denote the many-one reducibility relationship as $\mathrm{A} \leq_m \mathrm{B}$. In complexity theory, we usually pose additional restrictions on the complexity of r. $\mathrm{A} \leq_m \mathrm{B}$ means that the hardness of A does not exceed

the hardness of B.

In the setting of non-membership $g \notin \mathbf{CC}(t)$, we are not comparing two problems, but comparing one problem g with a set of problems in $\mathbf{CC}(t)$. To describe this relationship, we introduce the notion of universal reductions.

Definition 10. *Given a function $g \in \mathbf{TF_1}$, a partial function $r : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ is an unlabeled universal reduction from $\mathbf{CC}(t)$ to g if and only if*

$$\forall i \in \mathrm{UR}(t), \forall x \in \mathbb{N}, \phi_i(x) = g(r(i, x));$$

a partial function $r : \mathbb{N}^3 \rightharpoonup \mathbb{N}$ is a labeled universal reduction from $\mathbf{CC}(t)$ to g if and only if

$$\forall \langle i, k \rangle \in \mathrm{LR}(t), \forall x \in \mathbb{N}, \phi_i(x) = g(r(i, k, x)).$$

In the case of time complexity of Turing machines, the Cook-Levin theorem [55, 56, 18] is typically proved by a labeled universal reduction $r(i, k, x)$ from $\mathbf{NP}$ (and thus also $\mathbf{P}$) to SAT. The time complexity of that r is polynomially-bounded with respect to $\|x\|$ for any fixed i and k, but may not be polynomially-bounded with respect to $\|\langle i, k \rangle\|$.

Note that in Definition 10, the function g can be generalized to non-total or non-computable functions as long as the equations hold for programs in $\mathbf{CC}(t)$. For this generalization, we also implicitly use the following simple fact throughout this chapter.

Fact 4. *Let f, r, g be partial functions. If $f \circ r$ is a universal reduction from $\mathbf{CC}(t)$ to g, then r is a universal reduction from $\mathbf{CC}(t)$ to $g \circ f$.*

4.5 From Witness Functions to Universal Reductions

Apparently, witness functions only require *one* difference point between every $\phi_i, i \in \mathbf{CC}(t)$ and g, while universal reductions require g to "encode" the information of *every* point for every $\phi_i, i \in \mathbf{CC}(t)$. However, due to compositions of programs, the existence of

witness functions for common complexity classes naturally leads to the existence of universal reductions. On the other hand, the complexity relations between witness functions and universal reductions depend on whether the representations of $\mathbf{CC}(t)$ is unlabeled or labeled. In this section we prove two theorems converting witness functions to universal reductions, for unlabeled and labeled representations, respectively. We can then deduce properties of witness functions from properties of universal reductions.

4.5.1 The Unlabeled Case

Definition 11. *A complexity class $\mathbf{CC}(t)$ is constant-closed if and only if $\forall c \in \mathbb{N}, \exists k \in \mathbb{N}, \forall x \in \mathbb{N}, t(k, x) \geq c$.*

Almost all realistic complexity classes are constant-closed, meaning that they include all constant complexity computable functions.

Theorem 9 (Witness Functions to Universal Reductions, Unlabeled)**.** *Given a constant-closed complexity class $\mathbf{CC}(t)$ and an unlabeled witness function w for $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$, there is a linear complexity computable function w such that $w \circ \mathsf{w}$ is an unlabeled universal reduction from any complexity class $\mathbf{CC}(t')$ to g. In particular, we can choose $t' = t$.*

Proof. Consider the following computable function with index j

$$\forall i \in \mathrm{UR}(t'), \forall x, y \in \mathbb{N}, \phi_j(i, x, y) = \neg\phi_u(i, x) = \neg\phi_i(x).$$

According to the S-m-n property, $\phi_{s_2^1(j,i,x)}(y) = \phi_j(i, x, y) = \neg\phi_i(x)$. Because $\mathbf{CC}(t)$ is constant-closed and $s_2^1(j, i, x)$ is a program of constant complexity independent of y (note that $\phi_i(x) \downarrow$ because $i \in \mathrm{UR}(t')$), we have

$$s_2^1(j, i, x) \in \mathrm{UR}(t).$$

By our assumption, the unlabeled witness function w works on it:

$$\phi_{s_2^1(j,i,x)}(w(s_2^1(j,i,x))) = \neg g(w(s_2^1(j,i,x))).$$

Overall we have

$$\begin{aligned} \forall i \in \mathrm{UR}(t'), \forall x \in \mathbb{N}, \phi_i(x) &= \neg\phi_{s_2^1(j,i,x)}(w(s_2^1(j,i,x))) \\ &= g(w(s_2^1(j,i,x)). \end{aligned}$$

Letting $\mathsf{w}(i,x) = s_2^1(j,i,x)$, we have

$$\forall i \in \mathrm{UR}(t'), \forall x \in \mathbb{N}, \phi_i(x) = g((w \circ \mathsf{w})(i,x)).$$

w is linear complexity computable because s_2^1 is, according to Axiom 2. □

In Section 4.6, we will define constructible functions (as complexity upper bounds) and prove the generalized hierarchy theorem. With that in mind, Theorem 9 alludes that if we restrict our discussion to constructible complexity upper bounds, then for $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$ where $\mathbf{CC}(t)$ is constant-closed, there is no upper bound for any unlabeled witness function, including the naive witness function implemented in Procedure Procedure 1. Indeed, since g's complexity is fixed, choosing higher and higher t' forces w to have higher and higher complexity lower bounds.

Moreover, in the unlabeled case, we have the following corollary that does not need the notion of "constructible functions".

Corollary 2. *For any constant-closed complexity class $\mathbf{CC}(t)$ and any $g \in \mathbf{TF_1} \setminus \mathbf{CC}(t)$, for any 1-ary computable function $v \in \mathbf{TF_1}$, there exists a linear complexity computable many-one reduction from v to $g \circ w$.*

Proof. Let $\phi_{i_v} = v$ and construct $\phi_l(x,y) = \phi_{i_v}(x)$. By the S-m-n property $\phi_{s_1^1(l,x)}(y) =$

$\phi_l(x, y) = \phi_{i_v}(x)$. Since $s_1^1(l, x) \in \mathrm{UR}(t)$, according to Theorem 9 with $t' = t$ we have

$$\forall x \in \mathbb{N}, v(x) = \phi_{s_1^1(l,x)}(0) = g((w \circ \mathsf{w})(s_1^1(l, x), 0)) = (g \circ w)(\mathsf{w}(s_1^1(l, x), 0)).$$

Here $\lambda x.\mathsf{w}(s_1^1(l, x), 0)$ is the desired many-one reduction, which is linear complexity computable by Axiom 1-Axiom 3. □

4.5.2 The Labeled Case

To prove the labeled version of Theorem 9, we need the following axiom, which says the resulting parameterized program must be larger than every argument in $x_1, \ldots, x_m$ when interpreted as natural numbers. This axiom holds on reasonable programming systems since the s_m^n function hardcodes the $x_1, \ldots, x_m$ values into the final program.

Axiom 7 (S-m-n Size)**.** *For any $m, n \in \mathbb{N}$, the s_m^n function satisfies*

$$s_m^n(i, \vec{x}) \geq \max\{x_1, \ldots, x_m\}$$

where $x \in \mathbb{N}^m$.

We also need some assumptions on the witness function w and the complexity bound t for proving the main theorem in this section. These conditions hold in common complexity classes.

First, if $g \notin \mathbf{CC}(t)$, then in typical scenarios, a witness function w has infinitely many choices of inputs for every candidate program p in $\mathbf{CC}(t)$. That is because if there are only finitely many different points between p and g, we can hardcode those different points in p to let p compute g, which is a contradiction for complexity classes closed under finite modifications. For this reason, we can consider witness functions such that $w(i, k) \geq i$.

Second, we consider a specific class of complexity functions that are *robust*.

Definition 12. *A complexity function $t(k, x)$ is robust if and only if it is non-decreasing*

with respect to k, x, and satisfies

$$\begin{aligned} \forall c, k, x \in \mathbb{N}, t(ck + c, x) &\geq ct(k, x) + c; \\ \forall k, x \in \mathbb{N}, t(k, x) &\geq \|x\|. \end{aligned}$$

A complexity class is robust if and only if it can be defined as $\mathbf{CC}(t)$ where t is a robust complexity function.

In other words, a robust complexity function is non-decreasing, uses the label k as (at least) both additive and multiplicative factors, and is at least linear with respect to the input length. Note that many complexity classes can be re-defined using robust complexity functions, in which cases k might play slightly different roles. For example, $\mathbf{P}$ can be re-defined as $\mathbf{CC}(t_p)$ where $t_p(k, x) = \lambda kx.k + (k + 1)\|x\|^{k+1}$. It is obvious that $t_p(k, x)$ is non-decreasing with respect to k and x, $t_p(k, x) \geq \|x\|$, and $t_p(0k+0, x) \geq 0t_p(k, x)+0 = 0$. For $c \geq 1$, we have

$$\begin{aligned} t_p(ck + c, x) &= (ck + c) + (ck + c + 1)\|x\|^{ck+c+1} \\ &\geq c + ck + c(k + 1)\|x\|^{k+1} \\ &= c + c(t_p(k, x)). \end{aligned}$$

Thus $\mathbf{P}$ is a robust complexity class. Robust complexity classes are constant-closed, because $\forall c \in \mathbb{N}, t(2c, x) = t(c \cdot 1 + c, x) \geq ct(1, x) + c \geq c$.

Recall that $\Gamma(\|i\| + \|\vec{x}\|) + \gamma(i)J(\Phi_i(\vec{x}))$ is the interpretation overhead described in Axiom 5. We use the (fixed) functions γ and J in the following theorem.

Theorem 10 (Witness Functions to Universal Reductions, Labeled)**.** *Given a robust complexity class $\mathbf{CC}(t)$ where t is a robust complexity function, suppose w is a labeled witness function for $g \notin \mathbf{CC}(t)$ such that $w(i, k) \geq i$. For any complexity function t' such that $\forall k, x \in \mathbb{N}, J(t'(k, x)) \leq t(k, x)$, there exist a linear complexity computable function w_0*

and a constant C such that

$$w \circ \lambda ikx.\langle \mathsf{w}_0(i,k,x), C(\|i\| + \|x\| + 1)(\gamma(i)+1)(k+1)\rangle$$

is a labeled universal reduction from $\mathbf{CC}(t')$ to g.

Remark 1. *Before giving the proof, we first explain this result in terms of complexity. The term $C(\|i\|+\|x\|+1)(\gamma(i)+1)(k+1)$ is a natural number represented in binary form. We estimate the time needed to compute it on common computation models such as single-tape Turing machines.*

First, the interpretation overhead coefficient $\gamma(i)$ can be made $O(\|i\|^2)$ on single-tape Turing machines, because we can divide the tape into two tracks, use the first track to contain machine i's tape, and use the second track to contain machine i's description and states (which are moved along with the simulation). One step of such simulation and movement takes $O(\|i\|^2)$ time.

Let $n = \|\langle i,k,x\rangle\|$. Assuming $\gamma(i) = O(\|i\|^d)$, the natural number $C(\|i\| + \|x\| + 1)(\gamma(i) + 1)(k + 1)$ in Theorem 10 can be computed by multiplying three numbers of (binary) lengths $O(\log n)$, $O(\log n)$, $O(n)$, respectively. Note that the value of $\|i\|$ is of order $O(n)$, so its binary representation length is of order $O(\log n)$. The other two cases can be similarly analyzed. Thus under common computation models, the multiplication takes $O(n(\log n)^2)$ complexity with respect to the input length, which is close to linear complexity.

Proof. Consider the partial computable function

$$f(i,x,y) = \begin{cases} \neg\phi_u(i,x) & \mathsf{if}\ y \geq x \\ 0 & \mathsf{else} \end{cases}.$$

Let

$$\begin{aligned}\phi_a(i,x,y) &= y \geq x,\\ \phi_b(x) &= \neg x,\\ \phi_c() &= 0.\end{aligned}$$

Note that a, b, c are all fixed programs. According to our axioms, we can define

$$\phi_{b_3(a,h_1^3(b,r_2^1(u)),r_0^3(c))} = f(i,x,y).$$

Let $e = b_3(a, h_1^3(b, r_2^1(u)), r_0^3(c))$, which is a fixed 3-ary program.

For $\langle i,k\rangle \in \mathrm{LR}(t')$ and $y \geq x$, we have

$$\begin{aligned}
& \lambda ixy.\Phi_{s_2^1(e,i,x)}(y)\\
=\ & O(\lambda ixy.\|i\| + \|x\| + \Phi_e(i,x,y))\\
=\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \Phi_{h_1^3(b,r_2^1(u))}(i,x,y) + \Phi_a(i,x,y))\\
=\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \max\{\Phi_b(0),\Phi_b(1)\} + \Phi_{r_2^1(u)}(i,x,y) + \Phi_a(i,x,y))\\
=\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \Phi_u(i,x))\\
=\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \gamma(i)J(\Phi_i(x)))\\
\leq\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \gamma(i)J(t'(k,x)))\\
\leq\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \gamma(i)t(k,x))\\
\leq\ & O(\lambda ixy.\|i\| + \|x\| + \|y\| + \gamma(i)t(k,y))
\end{aligned}$$

where the last inequality utilizes the fact that t is non-decreasing. An important observation is that the last three $\leq$ signs does not introduce new constants hidden by the big-O notation. In other words, those hidden constants do not depend on k.

For $\langle i, k\rangle \in \mathrm{LR}(t')$ and $y < x$, we have

$$\begin{aligned}
& \lambda ixy.\Phi_{s_2^1(e,i,x)}(y) \\
= \;& O(\lambda ixy.\|i\| + \|x\| + \Phi_e(i,x,y)) \\
= \;& O(\lambda ixy.\|i\| + \|x\| + \|y\| + \Phi_{r_0^3(c)}(i,x,y) + \Phi_a(i,x,y)) \\
= \;& O(\lambda ixy.\|i\| + \|x\| + \|y\| + \Phi_c()) \\
= \;& O(\lambda ixy.\|i\| + \|x\| + \|y\|).
\end{aligned}$$

Overall, there exists a constant $C \in \mathbb{N}$ independent of i, k, x, y such that

$$\begin{aligned}
\lambda ixy.\Phi_{s_2^1(e,i,x)}(y) &\leq O(\lambda ixy.\|i\| + \|x\| + \|y\| + \gamma(i)t(k,y)) \\
&\leq \lambda ixy.C(\|i\| + \|x\| + 1) + C(\gamma(i)+1)t(k,y)
\end{aligned}$$

where the last inequality utilizes the fact $t(k, y) \geq \|y\|$.

It is necessary to compute a k' satisfying $\forall y \in \mathbb{N}, \Phi_{s_2^1(e,i,x)}(y) \leq t(k', y)$ before invoking w on $\langle s_2^1(e, i, x), k'\rangle$. Let $D = C(\gamma(i) + \|i\| + \|x\| + 1)$. Because t is robust, we have

$$t(D(k+1), y) = t(Dk + D, y) \geq Dt(k, y) + D \geq \Phi_{s_2^1(e,i,x)}(y).$$

Thus we can choose any $k' \geq D(k+1) = C(\gamma(i) + \|i\| + \|x\| + 1)(k+1)$. Let $k' = C(\|i\| + \|x\| + 1)(\gamma(i)+1)(k+1)$ and define

$$\begin{aligned}
\mathsf{w}_0(i,k,x) &= s_2^1(e,i,x), \\
\mathsf{w}(i,k,x) &= \langle \mathsf{w}_0(i,k,x), k'\rangle.
\end{aligned}$$

w_0 is linear complexity computable. Applying the labeled witness function on $\langle \mathsf{w}_0(i,k,x), k'\rangle$ gives

$$\phi_{s_2^1(e,i,x)}(w(s_2^1(e,i,x), k')) = \neg g(w(s_2^1(e,i,x), k')).$$

On the other hand, because we assume $\forall i, k \in \mathbb{N}, w(i,k) \geq i$ and Axiom 7, the first branch

in $\phi_{s_2^1(e,i,x)}(y) = f(i,x,y)$ is taken:

$$w(s_2^1(e,i,x),k') \geq s_2^1(e,i,x) \geq x \Rightarrow \phi_{s_2^1(e,i,x)}(w(s_2^1(e,i,x),k')) = \neg\phi_u(i,x) = \neg\phi_i(x).$$

Overall we have

$$\begin{aligned}\forall\langle i,k\rangle \in \mathrm{LR}(t'), \forall x \in \mathbb{N}, \phi_i(x) &= \neg\phi_{s_2^1(e,i,x)}(w(s_2^1(e,i,x),k')) \\ &= g(w(s_2^1(e,i,x),k')) \\ &= g((w \circ \mathsf{w})(i,k,x)).\end{aligned}$$

□

Example 4. *Consider the time complexity of single-tape Turing machines where we can set $J(x) = x$. The condition $\forall k, x \in \mathbb{N}, J(t'(k,x)) \leq t(k,x)$ in Theorem 10 is slightly tricky: we may need to adjust the k in common complexity classes in order to let this condition be satisfied. For any fixed $d \geq 1$, consider the following definitions of $\mathbf{TIME}(n^d)$ and $\mathbf{P}$.*

$$\begin{aligned}\mathbf{TIME}(n^d) &= \mathbf{CC}(t'(k,x)) &= \mathbf{CC}(\lambda kx.(k+(k+1)\|x\|)^d), \\ \mathbf{P} &= \mathbf{CC}(t(k,x)) &= \mathbf{CC}(\lambda kx.(k+(k+1)\|x\|)^{k+d}).\end{aligned}$$

In the labeled representation $LR(t)$ for $\mathbf{P}$, an element $\langle i,k\rangle$ means "i is a program whose execution time is upper-bounded by $(k+(k+1)\|x\|)^{k+2d}$". Obviously, if we know some other polynomial upper bound for i, we can find such a k as well.

First of all, $t(k,x)$ is a robust complexity function, because it is non-decreasing with respect to both k and x, it is lower-bounded by $\|x\|$, and it is easy to verify that $\forall c \geq 1$,

$$\begin{aligned}t(ck+c,x) &= (ck+c+(ck+c+1)\|x\|)^{ck+c+d} \\ &\geq c+(ck+c(k+1)\|x\|)^{ck+c+d} \\ &\geq c+c(k+(k+1)\|x\|)^{k+d} \\ &= c+ct(k,x).\end{aligned}$$

Second, we have

$$J(t'(k,x)) = t'(k,x) \leq t(k,x).$$

so Theorem 10's conditions are satisfied.

Suppose $\mathrm{SAT} \notin \mathbf{P}$ *and let* w *be a labeled witness function for* $\mathrm{SAT} \notin \mathbf{CC}(t) = \mathbf{P}$ *such that* $w(i,k) \geq i$*. According to Theorem 10, there exist a linear complexity computable function* w_0*, such that*

$$w \circ \lambda ikx.\langle \mathsf{w}_0(i,k,x), C(\|i\| + \|x\| + 1)(\gamma(i)+1)(k+1)\rangle$$

is a labeled universal reduction from $\mathbf{CC}(t') = \mathbf{TIME}(n^d)$ *to* SAT*. Fixing* $\langle i,k\rangle \in \mathbf{CC}(t')$*, the function* $\lambda ikx.\langle \mathsf{w}_0(i,k,x), C(\|i\|+\|x\|+1)(\gamma(i)+1)(k+1)\rangle$ *is computable in linear time with respect to* $\|x\|$*. So we get a linear time many-one reduction from* i *to* $\mathrm{SAT} \circ w$*. This is very different from the Cook-Levin theorem's reduction from* $\mathbf{NTIME}(n^d)$*, and thus also* $\mathbf{TIME}(n^d)$*, to* SAT*, where the* $O(n^d)$ *length execution trace is expanded to a Boolean formula, resulting in a reduction of (unoptimized)* $O(p(n)^3 \log(p(n)))$ *time or (optimized)* $O(p(n)\log(p(n)))$ *[57] time where* $p(n)$ *is a* d*-degree polynomial. Since our reduction from* i *to* $\mathrm{SAT} \circ w$ *is linear, where only limited transformations can be done in such limited time, this alludes that* $\mathrm{SAT} \circ w$ *directly encodes* i*'s semantics without expanding its execution trace.*

Overall, as discussed in Remark Remark 1, the universal reduction is of $O(n(\log n)^2)$ *time under common computation models. This provides further allusions to the structure of* $\mathrm{SAT} \circ w$*. While the previous paragraph shows that* $\mathrm{SAT} \circ w$ *directly encodes the semantics of each specific program in* $\mathbf{TIME}(n^d)$ *without expanding its execution trace, the overall complexity of the universal reduction shows that* $\mathrm{SAT} \circ w$ *directly encodes every program in* $\mathbf{TIME}(n^d)$*. This makes* $\mathrm{SAT} \circ w$ *be similar to the artificial problems constructed in time / space hierarchy theorems [17, 18], which are themselves defined as sets of programs / inputs.*

4.6 The Generalized Hierarchy Theorem

In this section we prove a generalized hierarchy theorem, of which the time / space hierarchy theorems [17, 18] on Turing machines are special cases. We will show that the proof implicitly includes a labeled witness function, which means witness functions are concrete entities existing in well-known proofs.

First, we introduce the following definition, which is a natural generalization of time / space constructible functions on Turing machines.

Definition 13. *A function $t \in \mathbf{TF}$ is constructible if and only if*

$$\exists j \in \mathbb{N}, \phi_j = t \wedge \Phi_j = O(t).$$

Constructible functions are suitable for being complexity upper bounds in hierarchy theorems because of the following axiom.

Axiom 8 (Bounded Execution)**.** *For any $n, i, v \in \mathbb{N}$ and any constructible function t, there exists $b \in \mathbb{N}$ such that*

$$\left(\forall \vec{x} \in \mathbb{N}^n, \phi_b(\vec{x}) = \begin{cases} 1 & \text{if } \phi_i(\vec{x}) = v \wedge \Phi_i(\vec{x}) \leq t(\vec{x}) \\ 0 & \text{otherwise} \end{cases} \right) \wedge \Phi_b = O(\lambda \vec{x}.t(\vec{x}) + \|\vec{x}\| + K(t(\vec{x})))$$

where K is a total, non-decreasing function.

For time complexity under realistic computation models such as single tape Turing machines, the program b on input $\vec{x}$ can first compute $T = t(\vec{x})$ on one track of the tape and then execute ϕ_i on $\vec{x}$ on the other track while moving and decreasing the step counter T. Updating and moving the step counter adds a logarithmic overhead, resulting in $K(y) = y \log y$. Finally b can effectively check whether $\phi_i(\vec{x})$ halts within T steps with the result v, all in constant time. Note that ϕ_b does not take i as input. This means b can just hardcode i's transition table instead of implementing a logic to simulate any given Turing machine i.

This is why the bound of Φ_b does not need terms depending on i, which is different from Axiom 5.

The following axiom states that any program can be padded with an arbitrarily large redundant code fragment, such that the resulting program still has the same extensional behavior and the same complexity, both on the bare metal computation model and on the interpreter ϕ_u. As an example in real programming languages, the interpreter can ignore some obviously redundant texts in the program such as comments.

Axiom 9 (Padding)**.** *There exists a* 2*-ary computable function* p *such that for any* $i, n \in \mathbb{N}$*.*

$$p(i,n) > n \wedge \phi_i = \phi_{p(i,n)} \wedge \Phi_i = \Phi_{p(i,n)} \wedge \lambda x.\Phi_u(i,x) = \lambda x.\Phi_u(p(i,n),x).$$

Theorem 11 (Generalized Hierarchy Theorem)**.** *If* t_1*,* t_2 *are constructible functions such that*

$$\forall c, \exists x_0, \forall x > x_0, c + c\|x\| + cJ(c + ct_1(x))) \leq t_2(x),$$

then we have

$$\mathbf{CC}(\lambda kx.k + kt_1(x)) \subsetneq \mathbf{CC}(\lambda kx.k + k(t_2(x) + \|x\| + K(t_2(x)))).$$

Proof. Let $\phi_e(x) = x$ and $\phi_{h_2^1(u,e,e)} = \phi_u(x,x)$. Let $j = h_2^1(u,e,e)$. According to our axioms we have

$$\lambda x.\Phi_j(x) = O(\lambda x.\Phi_u(x,x) + \Phi_e(x)) = O(\lambda x.\Phi_u(x,x) + \|x\|).$$

That means there exists C_0 such that

$$\Phi_j(x) \leq C_0 + C_0(\Phi_u(x,x) + \|x\|).$$

According to Axiom 8, there exists $b \in \mathbb{N}$ such that

$$\phi_b(x) = \begin{cases} 1 & \text{if } \phi_j(x) = 0 \wedge \Phi_j(x) \leq t_2(x) \\ 0 & \text{otherwise} \end{cases} \quad \wedge \Phi_b = O(\lambda x.t_2(x) + \|x\| + K(t_2(x))).$$

Obviously $\phi_b \in \mathbf{CC}(\lambda kx.k + k(t_2(x) + \|x\| + K(t_2(x))))$. Suppose for contradiction that $\phi_b \in \mathbf{CC}(\lambda kx.k + kt_1(x))$. Then there exists $d \in \mathbb{N}$ such that

$$\phi_d = \phi_b \wedge \Phi_d = O(t_1).$$

Let C_1 be the constant such that $\Phi_d(x) \leq C_1 + C_1 t_1(x)$. For this d, according to Axiom 5 we have

$$\Phi_u(d, x) \leq \Gamma(\|d\| + \|x\|) + \gamma(d)J(\Phi_d(x)) \leq \Gamma(\|d\| + \|x\|) + \gamma(d)J(C_1 + C_1 t_1(x)).$$

According to the theorem's premise of the relation between t_1 and t_2, we can choose a large enough n such that

$$\begin{aligned} & \forall x > n, C_0 + C_0(\Phi_u(d, x) + \|x\|) \\ \leq\ & C_0 + C_0(\Gamma(\|d\| + \|x\|) + \gamma(d)J(C_1 + C_1 t_1(x)) + \|x\|) \\ =\ & (C_0 + C_0\Gamma\|d\|) + (C_0\Gamma + C_0)\|x\| + C_0\gamma(d)J(C_1 + C_1 t_1(x)) \\ \leq\ & t_2(x). \end{aligned}$$

According to Axiom 9, let $d_1 = p(d, n) > n$ where

$$\phi_d = \phi_{d_1} \wedge \Phi_d = \Phi_{d_1} \wedge \lambda x.\Phi_u(d, x) = \lambda x.\Phi_u(d_1, x).$$

In this case, we have

$$\Phi_j(d_1) \leq C_0 + C_0(\Phi_u(d_1, d_1) + \|d_1\|) = C_0 + C_0(\Phi_u(d, d_1) + \|d_1\|) \leq t_2(d_1).$$

On the other hand, according to the assumption $\phi_{d_1} = \phi_d = \phi_b$, we have $\phi_{d_1}(d_1) = 1 \Leftrightarrow \phi_{d_1}(d_1) = 0$, which is a contradiction. □

In Theorem 11, $\phi_{d_1} = \phi_d$ always holds, but with the assumption $\phi_d(d_1) = \phi_b(d_1)$ we can derive a contradiction. This means $\phi_d(d_1) \neq \phi_b(d_1)$, i.e. the function $d \mapsto d_1$ is a witness function for $\phi_b \notin \mathbf{CC}(\lambda kx.k + kt_1(x))$. More precisely, d_1 can be defined as

$$d_1 = p(d, \text{any}_n\{\forall x > n, (C_0{+}C_0\Gamma\|d\|){+}(C_0\Gamma{+}C_0)\|x\|{+}C_0\gamma(d)J(C_1{+}C_1t_1(x)) \leq t_2(x)\})$$

where "any" means any such n constitutes a valid definition of d_1. The constants C_0, Γ and functions γ, t_1, t_2 are all independent of d, but C_1 could depend on d. Indeed, C_1 is an additive and multiplicative factor in $\Phi_d = O(t_1) = \lambda x.C_1 + C_1t_1(x)$. This implies that the witness function w is labeled, i.e., it needs k as a parameter such that $w(d, k) = d_1$.

4.7 Relations to the Relativization Barrier

In 1975, Baker, Gill, and Solovay [12] constructed oracles A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$. This shows that any technique being capable of resolving the $\mathbf{P}$ versus $\mathbf{NP}$ problem must not generalize to all oracles. Since many existing diagonalization proofs generalize to all oracles [12, 58, 59], this result became the well-known *relativization barrier* in complexity theory, showing the limitation of techniques *relativizing* to arbitrary oracles. The relativization barrier was frequently mentioned as evidence showing the limits of diagonalization [12, 58, 59, 60], while because of the lack of formal definitions of "diagonalization", the exact relation between "relativizing techniques" and "diagonalization" is unclear. Aaronson [58] described diagonalization and relativization barrier as:

> The magic of diagonalization, self-reference, and counting arguments is how abstract and general they are [...] the price of generality is that the logical techniques are extremely limited in scope.

Nevertheless, Kozen [14] proposed a formalization of diagonalization and claims "if $\mathbf{P} \neq \mathbf{NP}$ is provable at all, then it is provable by diagonalization". Combining with Baker, Gill, and Solovay's result, Kozen claimed:

> [. . .] there must exist diagonalizations which do not relativize.

Fortnow [61] commented on Kozen's result as:

> I believe this result says more about the difficulty of exactly formalizing the notion of a "diagonalization proof" than of actually arguing the diagonalization technique is the only technique we have for class separation.

Since witness functions exist as long as the separation $X \notin \mathbf{C}$ is true (on the standard model of arithmetic), our results are not restricted to relativizing techniques. Indeed, Kozen's result [14] is essentially saying the existence of witness functions as long as $X \notin \mathbf{C}$ is provable (and is thus true assuming the soundness of the proof system). So if witness functions are regarded as a general definition of "diagonalization", then it is "the" technique that we must (explicitly or implicitly) resort to for class separations.

4.8 Related Work

Kozen's work [14] concluded that if $\mathbf{P} \neq \mathbf{NP}$ is provable at all, then it is provable by diagonalization. That argument was based on the existence of computable witness functions, which is equivalent to the separation itself. Notably, Kozen's work [14] also concluded a preliminary form of our main constructions: if $g \notin \mathcal{C}$ and if w is a witness function for this fact, then the universal function of $\mathcal{C}$ is reducible to the pair $\langle g, w \rangle$. However, the detailed complexity of w and the difference between unlabeled and labeled representations were not discussed. Subsequent work by Joseph and Young [13] discussed the complexity of witness functions in the setting of $\mathbf{P}$, $\mathbf{NP}$, $\mathbf{coNP}$, but didn't discuss it in a more general setting nor discuss the fine-grained complexity of the witness functions.

Nash, Impagliazzo, and Remmel [62] defined strong and weak diagonalizations and argued that those two notions are indeed different. For separating two complexity classes $\mathcal{A} \supsetneq \mathcal{B}$, their definition of strong diagonalization require the "weak universal language" to be in $\mathcal{A}$. In our case, however, we do not require the universal reduction to belong to any complexity class; instead, we only show that universal functions can be expressed by the witness function, whose complexity could be unbounded as shown in Theorem 9.

Witness functions or similar forms had also been considered in computability settings, such as productive functions for computably enumerable subsets [15] and witness functions for computable subsets [63]. In our complexity settings, however, we need to consider the complexity of sets and functions and thus we introduce Blum-like axioms [50] to develop our theorems in a general way.

4.9 Chapter Conclusion

We studied the properties of witness functions in the complexity theory setting. We showed that labeled witness functions are the ones implicitly used in real complexity theoretic proofs, and relating these witness functions to universal reductions can shed light on the possible ways to separate complexity classes, including long-standing open problems in complexity theory.

CHAPTER 5

EXAMPLE: MUTUAL REFINEMENTS OF CONTEXT-FREE LANGUAGE REACHABILITY

5.1 Introduction

Context-free language reachability (CFL-reachability) is arguably the best-known graph reachability framework in program analysis [64, 65, 66, 67, 68, 69]. Typically, the framework consists of a frontend and a backend, where the frontend constructs a graph from the source code, and the backend runs CFL-reachability on the graph to obtain properties of the source code [70]. The graphs and grammars depend on specific analyses, but CFL-reachability has a dynamic programming style (sub)cubic-time algorithm [71, 70, 72, 73] for arbitrary graphs and grammars. Faster algorithms exist on special cases [74, 75, 76].

However, due to the inherent hardness of program analysis, CFL-reachability may not be able to model the exact formulation of the problem [4, 77]. A typical example is the interleaved-Dyck-reachability formulation [4, 78], which is widely used to simultaneously model function calls/returns [66, 4], field reads/writes [79, 80], locks/unlocks [81], etc. The interleaved-Dyck language is not context-free [82], and the corresponding graph reachability problem is undecidable [4]. In practice, CFL-reachability can over-approximate computationally hard language reachability problems [4]: the idea is to design a context-free language C that over-approximates the non-context-free language L (meaning that C contains more strings than L).

For a computationally hard L-reachability problem, different CFL-reachability-based approaches over-approximate the solution from different angles. We can straightforwardly intersect the results to achieve better precision. Synchronized pushdown systems [83] essentially employ this idea. The linear conjunctive language reachability work [77] also

shows this straightforward intersection can improve precision. However, in this case, different CFL-reachability instances are executed independently. On the other hand, CFLs are not closed under intersection [84, 82], so in general, we cannot intersect different CFLs to obtain a new CFL for CFL-reachability. For example, the interleaved-Dyck language [78, 77], which is not a CFL, is the intersection of two or more CFLs.

This chapter proposes a more synergistic way to "intersect" multiple CFL-reachability-based over-approximations. Specifically, typical CFL-reachability algorithms are of dynamic programming style, which generate "summary edges" from existing graph edges. Our key insight is that when those algorithms generate a summary edge, the existing edges directly contributing to the generation can be recorded. This augmented algorithm can eventually trace a set of original graph edges that contribute to the final reachability results. Therefore, when combining CFL-reachability-based over-approximations based on CFLs $C_1, C_2, \ldots, C_n$, we can run C_2-reachability on the contributing edges of C_1-reachability, as opposed to all edges in the original graph. This process can happen between different C_i's multiple times, which is called *mutual refinement*.

Is the execution order of different CFL-reachability over-approximations important for mutual refinement? We prove in Section 5.4 that given a set of CFL-reachability over-approximations, there exists a unique fix-point, and any order of executing different CFL-reachabilities will reach the fix-point. This is similar to the fix-point theorem [85] and chaotic-iteration algorithms [86, 87] in dataflow analysis. The soundness of the fix-point and the fact that it is at least as precise as the straightforward intersection are also proved in Section 5.4. As for the complexity, suppose the CFL is fixed and the number of vertices in the graph is n, then the time complexities of the standard CFL-reachability algorithm [71, 70, 72] and our augmented algorithm are both $\tilde{O}(n^3)$, and the space complexity is $\tilde{O}(n^2)$ for the standard algorithm and is $\tilde{O}(n^3)$ for our augmented algorithm.

For a fixed set of CFL-reachability over-approximations, the fix-point described in the previous paragraph is the best precision that can be achieved by the mutual refinement

method. However, mutual refinement is itself a decidable approximation of the undecidable problem. According to witnessability described in Chapter 3, as long as there is a many-one reduction from the halting problem to the undecidable problem here, there exists a computable witness function transforming mutual refinement to a counterexample on which mutual refinement is imprecise.

We conduct experiments on two applications: a taint analysis for Java programs obtained from Android apps [88], and a value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [89]. On average, compared with the straightforward intersection, mutual refinement achieves a 50.95% precision improvement (measured by the number of reachable pairs) with a $2.65\times$ time increase and a $3.23\times$ space increase on the taint analysis benchmarks, and achieves a 9.37% precision improvement with a $2.55\times$ time increase and a $2.22\times$ space increase on the value-flow analysis benchmarks.

The fast graph simplification algorithm [78] proposed by Li, Zhang, and Reps (abbreviated as the LZR algorithm) also simplify graphs, but the LZR algorithm only works for interleaved-Dyck-reachability while mutual refinement works for any L-reachability preserving CFL-reachability-based over-approximations, and the LZR algorithm is a preprocessing algorithm while mutual refinement is a complete solver. Our taint analysis experiment is interleaved-Dyck reachability, and thus we also evaluate mutual refinement on those graphs simplified by the LZR algorithm: LZR preprocessing can, on average, bring a further precision improvement of 3.27% and reduces the time/space consumption in certain cases. The value-flow analysis experiment is not interleaved-Dyck reachability, so the LZR algorithm is not applicable.

In summary, this chapter makes the following main contributions.

- We propose mutual refinement for combining different CFL-reachability over-approximations for hard formal language reachability problems.
- We prove the existence and uniqueness of the fix-point, the soundness, the precision guarantee (being at least as precise as the straightforward intersection), and time/s-

```
1  #include ...
2
3  class Pair {
4      int first, second;
5      Pair(int fi, int se) : first(fi), second(se) {}
6  }
7
8  int getFirst(Pair p1) { return p1.first; /* represented by ret1 */ }
9
10 int getSecond(Pair p2) { return p2.second; /* represented by ret2 */ }
11
12 int main() {
13     int s = getSecret();
14     Pair a(0, s), b(0, 0), t(0, 0);
15     if (getInput() == "first") {
16         int x = getFirst(a);
17         t.first = x;
18     } else {
19         send(getSecond(a));
20         int y = getSecond(b);
21         t.second = y;
22     }
23     ...
24 }
```

Figure 5.1: A taint analysis example for C++. The goal is to decide whether the value s can flow into t. The fact is that the value of s cannot flow into t.

pace complexities for mutual refinement.

- We evaluate mutual refinement on two program analysis applications. Experimental results show that mutual refinement can achieve better precision than the straightforward intersection with reasonable extra cost.

This chapter is organized as follows. Section 5.2 gives a motivating example. Section 5.3 reviews backgrounds and definitions. Section 5.4 presents mutual refinement and its properties. Section 5.5 gives experimental results. Section 5.6 presents discussions. Section 5.7 surveys related work, and Section 5.8 concludes.

5.2 Motivating Example

This section motivates mutual refinement using an example of context-sensitive and field-sensitive taint analysis. The analysis first generates a graph from the source code being analyzed, then the source-sink relation from the source code is reduced to the reachability of two vertices in the graph. This is an extended version of the taint analysis in the work

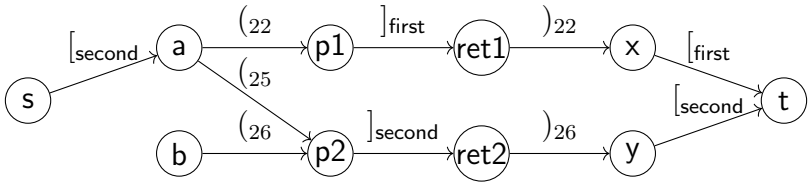


Figure 5.2: The taint analysis graph for Figure 5.1. Vertices are variables and edges model values flowing among variables: $i \xrightarrow{(_c} j$ represents that i flows into j via the function call at line c; $i \xrightarrow{)_c} j$ represents that i flows into j via the function return at line c; $i \xrightarrow{[_f} j$ represents that i flows into the f field of j; $i \xrightarrow{]_f} j$ represents that the f field of i flows into j.

of Huang *et al* [88]. The original analysis is based on interleaved-Dyck reachability, but our example is not. We compare mutual refinement with the straightforward intersection of two different CFL-reachability-based over-approximations.

Example Code. Figure 5.1 shows a C++ code snippet. The analysis goal is to decide whether the value of the variable s could flow into the variable t. Because t can only contain the first field of a or the second field of b, the value of s cannot flow into t.

Graph Reachability Formulation. We use vertices to represent variables and use edges to represent values flowing among variables (Figure 5.2). To achieve context-sensitivity and field-sensitivity, we use parenthesis-labeled edges for function calls/returns, and use bracket-labeled edges for field writes/reads.

The taint analysis decides whether the value of s can flow into t (including t's fields). The answer is "yes" if and only if there is a path whose edge labels can be concatenated to a string that is an interleaving of matched parentheses, matched brackets, and unmatched open brackets. Formally, given an alphabet Σ, we define the interleaving operator [78] $\odot : \Sigma^* \times \Sigma^* \to \mathcal{P}(\Sigma^*)$ as follows, where s, s_1, s_2 are strings and c_1, c_2 are characters.

$$
\begin{aligned}
\epsilon \odot s &= \{s\} \\
s \odot \epsilon &= \{s\} \\
c_1 s_1 \odot c_2 s_2 &= \{c_1 w \mid w \in (s_1 \odot c_2 s_2)\} \cup \{c_2 w \mid w \in (c_1 s_1 \odot s_2)\}.
\end{aligned}
$$

For the taint analysis, we are interested in L_T-reachability problem, where $L_T = \bigcup\{s_1 \odot$

$s_2 \mid s_1 \in P, s_2 \in B\}$ and CFLs P and B are defined as follows. Note that we use P or B to denote both the languages and the starting symbols in the grammars. L_T is not context-free (which is proved in Section 5.6). We also extend the definition of the interleaving operator $\odot$ to languages, and thus we write $L_T = P \odot B$.

$$P \rightarrow P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_m\ P\ )_m \mid \epsilon$$

$$B \rightarrow D\ B \mid [_1\ B \mid \ldots \mid [_n\ B \mid \epsilon.$$
$$D \rightarrow D\ D \mid [_1\ D\ ]_1 \mid \ldots \mid [_n\ D\ ]_n \mid \epsilon.$$

In Figure 5.2, there are only two possible paths from s to t. None of these paths satisfy our requirement, so the value of s cannot flow into t.

CFL-Reachability-Based Over-Approximations. We devise two CFLs C_P and C_B to over-approximate L_T. C_P considers only parentheses and treats brackets as empty symbols. C_B considers only brackets and treats parentheses as empty symbols. Both algorithms can be implemented using the well-known CFL-reachability algorithm [71, 70, 72]. In Figure 5.2, both C_P-reachability and C_B-reachability conclude that t is reachable from s. For example, for C_P, t is reachable from s via the path $\mathsf{s} \xrightarrow{[_{\mathsf{second}}} \mathsf{a} \xrightarrow{(_{22}} \mathsf{p1} \xrightarrow{]_{\mathsf{first}}} \mathsf{ret1} \xrightarrow{)_{22}} \mathsf{x} \xrightarrow{[_{\mathsf{first}}} \mathsf{t}$, and for C_B, t is reachable from s via the path $\mathsf{s} \xrightarrow{[_{\mathsf{second}}} \mathsf{a} \xrightarrow{(_{25}} \mathsf{p2} \xrightarrow{]_{\mathsf{second}}} \mathsf{ret2} \xrightarrow{)_{26}} \mathsf{y} \xrightarrow{[_{\mathsf{second}}} \mathsf{t}$. These two conclusions are both false positives.

Straightforward Intersection. One possible method to combine the above two over-approximations is to directly intersect their results, as synchronized pushdown system [83] and linear conjunctive language reachability [77] did. In Figure 5.2, however, this method still concludes that t is reachable from s, because the two algorithms both conclude that they are reachable.

Mutual Refinement. To further improve the precision, we first run C_B-reachability, which concludes that t is reachable from s, and the edges contributing to all reachable pairs can be shown in Figure 5.3 (parentheses are treated as empty symbols so edges labeled with parentheses are preserved). Now the graph has been simplified. Running C_P-reachability

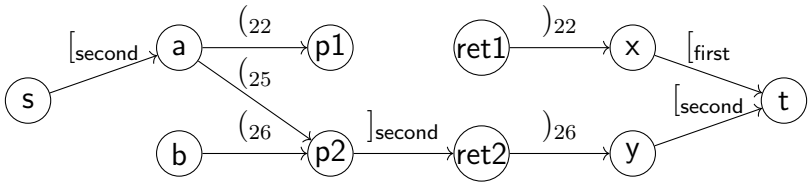


Figure 5.3: After running C_B-reachability and tracing only the edges contributing to its results, the graph is simplified. Subsequent execution of C_P-reachability can then conclude that t is not reachable from s.

on the graph in Figure 5.3 concludes that t is not reachable from s. Thus C_B-reachability "refines" the subsequent execution of C_P-reachability. This example shows that mutual refinement can achieve better precision compared with the straightforward intersection.

5.3 Preliminary

We review the standard dynamic programming style algorithm for CFL-reachability in Section 5.3.1. CFL-reachability can over-approximate other L-reachability problems.

5.3.1 CFL-Reachability

In L-reachability described in Section 2.2.1, when L is a context-free language, the problem is a CFL-reachability problem. CFL-reachability exhibits a popular dynamic programming style cubic-time algorithm [71, 70, 72], as shown in Algorithm 1. Given an instance $\langle C, (V, E)\rangle$ of (all-pairs) CFL-reachability problem, we call CFL-REACHABILITY$(\langle C, (V, E)\rangle)$ in Algorithm 1 to compute the results. The CFL $C = (\Sigma, N, P, S)$ contains the set of terminal symbols Σ, the set of non-terminal symbols N, the set of productions P, and the start symbol S. All productions are in the forms $X \to YZ$, $X \to Y$, and $X \to \epsilon$, where X, Y, and Z are terminal symbols or non-terminal symbols. Any context-free grammar can be transformed into this form [82]. Algorithm 1 works as follows.

1. Initially, all edges in the original graph are added to the worklist (line Line 3).

2. Then the productions with empty right-hand sides are applied where new edges are

Algorithm 1 The CFL-Reachability Algorithm

```
1:  function CFL-REACHABILITY(⟨C, (V, E)⟩)
2:      W ← emptyWorkList()
3:      W.addAll(E)
4:      for X → ϵ ∈ C do
5:          for v ∈ V do
6:              if X⟨v, v⟩ ∉ E then
7:                  add X⟨v, v⟩ to E and W
8:      while W.nonEmpty() do
9:          Y⟨i, j⟩ ← W.pop()
10:         for X → Y ∈ C do
11:             if X⟨i, j⟩ ∉ E then
12:                 add X⟨i, j⟩ to E and W
13:         for X → YZ ∈ C do
14:             for Z⟨j, k⟩ ∈ E do
15:                 if X⟨i, k⟩ ∉ E then
16:                     add X⟨i, k⟩ to E and W
17:         for X → ZY ∈ C do
18:             for Z⟨k, i⟩ ∈ E do
19:                 if X⟨k, j⟩ ∉ E then
20:                     add X⟨k, j⟩ to E and W
21:     return (V, E)
```

added to both the graph and the worklist (lines Line 4-Line 7).

3. After that, the algorithm removes an edge $Y\langle i, j\rangle$ (connecting vertices i, j with label Y) from the worklist, trying all productions with Y on the right-hand side, adding newly generated edges to both the graph and the worklist, and repeating this process until the worklist becomes empty (lines Line 8-Line 20).

4. Finally, the updated graph is returned as the result (line Line 21).

Algorithm 1 shows the process of generating new edges in the graph according to productions, which we call *production applications*. For example, if there is a production $X \to YZ$ and there are edges $Y\langle i, j\rangle$ and $Z\langle j, k\rangle$ already in the graph, we add a new edge $X\langle i, k\rangle$ to the graph via a production application. Note that when we pop an edge e out from the worklist and process it (lines Line 8-Line 20), depending on the previously pro-

cessed edges, some of e's adjacent edges might not be available in the graph yet, thus the current iteration of lines Line 8-Line 20 may not add all edges that can be generated from e via production applications. However, it is well-known that eventually, all possible edges will be added, and the order of popping out edges from the worklist does not matter.

Theorem 12 (Algorithm 1's Correctness)**.** *When Algorithm 1 terminates, all edges that can be generated by production applications will be in the graph.*

Proof. We prove it by contradiction. Suppose there is an edge e_n, which could be generated by a finite number of production applications, and which is not in the graph produced by Algorithm 1. It is obvious that e_n cannot be in the original edge set. So e_n could be obtained by finitely and positively many production applications. We can denote this process by a sequence $e_1, e_2, \ldots, e_n$, where for all $i \in \{1, 2, \ldots, n\}$, e_i is either in the original graph's edge set or is obtained by applying a production on a set of edges $\{e_{i_1}, e_{i_2}, \ldots, e_{i_{n_i}}\} \subseteq \{e_1, e_2, \ldots, e_{i-1}\}$. Suppose e_j is the first edge in $e_1, e_2, \ldots, e_n$ that is not in the graph produced by Algorithm 1. There must exist such an e_j because according to our assumption, at least e_n is not in the graph produced by Algorithm 1. Again, e_j cannot be in the original edge set. Suppose e_j can be obtained by applying a production on a set of edges $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\} \subseteq \{e_1, e_2, \ldots, e_{j-1}\}$. Since all edges in $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\}$ are in the graph produced by Algorithm 1, when the last edge was popped out, since all productions were tried, Algorithm 1 must add e_j to the graph since all edges in $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{n_j}}\}$ were available in the graph at that time. This is a contradiction. □

We briefly explain our notations for time/space complexity. To make the analysis rigorous, we should also consider the time/space complexity of handling numbers. For example, if there are n vertices in the graph and we use $0, 1, \ldots, n-1$ to represent the vertices, then the number $n-1$ itself requires memory of $O(\log n)$, and arithmetic operations on those numbers are not of constant complexity. To focus on dominating factors, instead of using the big-O notation, we use the $\tilde{O}$ notation [90] to hide logarithm factors: $\tilde{O}(f(n))$ repre-

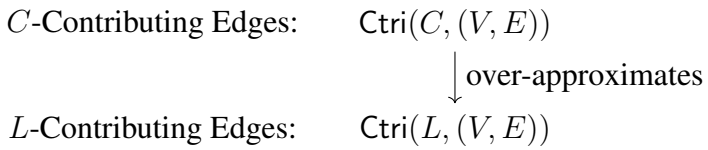


Figure 5.4: Two important concepts in mutual refinement. L is a formal language whose reachability problem is computationally hard, and C is a context-free language over-approximating L. The set of C-contributing edges $\mathsf{Ctri}(C, (V, E))$ over-approximates the set of L-contributing edges $\mathsf{Ctri}(L, (V, E))$.

sents $O(f(n)(\log n)^k)$ for some constant natural number k.

Complexity Analysis for Algorithm 1. Suppose the grammar of the CFL is fixed, adding one edge to the graph takes constant time, accessing each graph vertex's adjacent vertices takes linear time, and pushing/popping elements to/from the worklist takes constant time. There could be at most $\tilde{O}(|V|^2)$ edges popped out from the worklist at line Line 9, and for each edge popped out, there could be at most $\tilde{O}(|V|)$ adjacent edges to try in the main **while** loop. Thus the time complexity is $\tilde{O}(|V|^3)$. The space complexity is $\tilde{O}(|V|^2)$ since there could be at most $\tilde{O}(|V|^2)$ edges in the graph and in the worklist.

5.4 Mutual Refinement

This section formalizes mutual refinement. Specifically, Section 5.4.1 gives an overview; Section 5.4.2 presents the important definition of *contributing edges*; Section 5.4.3 presents the algorithm used as individual steps in mutual refinements; Section 5.4.4 presents the complete mutual refinement algorithm.

5.4.1 Overview

Suppose we have a set of CFL-reachability-based over-approximations using CFLs $C_1, C_2, \ldots, C_m$ for a computationally hard L-reachability problem. For an instance $\langle L, (V, E)\rangle$ of the problem, we first run C_1-reachability. Then we only keep edges that directly or indirectly participated in the construction of S_1 (the start symbol of C_1's grammar) edges. Then we

run C_2-reachability and only keep edges participated in the construction of S_2 edges. This process continues until we reach the fix-point: no more edges can be removed. The final reachability result is obtained by executing C_1, C_2, ..., C_m-reachability on the minimum graph and taking the intersection.

5.4.2 Contributing Edges

The key step of mutual refinement is to over-approximate the set of "useful" edges. This is achieved via formal language over-approximations.

Definition 14 (Formal Language Over-Approximation)**.** *Given two formal languages L_1 and L_2, L_1 over-approximates L_2 if and only if $L_1 \supseteq L_2$.*

Definition 15 (L-Contributing Edges)**.** *For a specific formal language L, given an instance $\langle L, (V, E)\rangle$ of the L-reachability problem, an edge $e \in E$ is an L-contributing edge for this instance if and only if there exists a pair of vertices $u, v \in V$, such that e is part of a finite path p connecting u and v and $R(p) \in L$. The set of such L-contributing edges is denoted as* $\mathsf{Ctri}(L, (V, E))$.

In certain undecidable L-reachability problems (*e.g.*, the interleaved-Dyck-reachability), it can be shown that computing the set $\mathsf{Ctri}(L, (V, E))$ is also undecidable in general [78]. So we need to approximate this set. Suppose we have a context-free language C over-approximating L, then it is straightforward to see that $\mathsf{Ctri}(L, (V, E)) \subseteq \mathsf{Ctri}(C, (V, E))$, because every L-path is also a C-path. Figure 5.4 summarizes the situation.

Example 5 (Contributing Edges)**.** *Consider the following example of interleaved-Dyck-reachability, where each string in the interleaved-Dyck language L is an interleaving of two strings from the following two CFLs, respectively.*

$$\begin{aligned} P &\to P\ P \mid (\ P\ ) \mid \epsilon \\ B &\to B\ B \mid [\ B\ ] \mid \epsilon. \end{aligned}$$

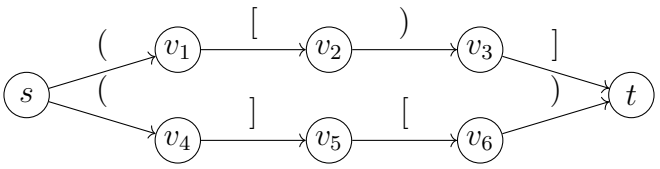


Figure 5.5: A Graph illustrating contributing edges.

We use the following context-free language C, which only considers matched parentheses and treats brackets as empty symbols, to over-approximate the interleaved-Dyck language.

$$C \quad \to C\ C \mid (\ C\ ) \mid [\ ] \mid \epsilon.$$

In the instance of interleaved-Dyck-reachability shown in Figure 5.5, the set of L-contributing edges is $\{s \xrightarrow{(} v_1, v_1 \xrightarrow{[} v_2, v_2 \xrightarrow{)} v_3, v_3 \xrightarrow{]} t\}$, while the set of C-contributing edges includes all edges in the original graph.

5.4.3 Tracing Algorithm

The set of CFL-contributing edges is computable via augmenting the standard CFL-reachability algorithm to trace the edges, which results in Algorithm 2. Given an instance $\langle C, (V, E)\rangle$ of a CFL-reachability problem, we first make the function call RECORD($\langle C, (V, E)\rangle$) to run the CFL-reachability algorithm and record the meta-information "metaInfo", where "metaInfo$[e]$" contains all edges that directly contributed to the construction of e. Notice that RECORD is almost the same as the standard CFL-reachability algorithm (Algorithm 1), except that we add the highlighted lines to record the meta-information. Then the original graph's C-contributing edges could be obtained by calling COLLECT(metaInfo, E), which recursively collects the contributing edges.[1]

[1] COLLECT can be implemented using either breadth-first-search or depth-first-search.

Algorithm 2 The Tracing Algorithm

```
1: function RECORD(⟨C, (V, E)⟩)
2:     metaInfo ← emptyMap()
3:     W ← emptyWorkList()
4:     W.addAll(E)
5:     for X → ϵ ∈ CFG do
6:         for v ∈ V do
7:             if X⟨v, v⟩ ∉ E then
8:                 add X⟨v, v⟩ to E and W
9:     while W.nonEmpty() do
10:        Y⟨i, j⟩ ← W.pop()
11:        for X → Y ∈ CFG do
12:            metaInfo[X⟨i, j⟩].add(Y⟨i, j⟩)
13:            if X⟨i, j⟩ ∉ E then
14:                add X⟨i, j⟩ to E and W
15:        for X → YZ ∈ CFG do
16:            for Z⟨j, k⟩ ∈ E do
17:                metaInfo[X⟨i, k⟩].add(Y⟨i, j⟩, Z⟨j, k⟩)
18:                if X⟨i, k⟩ ∉ E then
19:                    add X⟨i, k⟩ to E and W
20:        for X → ZY ∈ CFG do
21:            for Z⟨k, i⟩ ∈ E do
22:                metaInfo[X⟨k, j⟩].add(Z⟨k, i⟩, Y⟨i, j⟩)
23:                if X⟨k, j⟩ ∉ E then
24:                    add X⟨k, j⟩ to E and W
25:    return ((V, E), metaInfo)
26:
27: function COLLECT(((V, E), metaInfo))
28:    visited ← ∅
29:    con ← ∅
30:    function COLLECTDFS(e)
31:        if e ∉ visited then
32:            visited.add(e)
33:            if e's label is a terminal symbol then
34:                con.add(e)
35:            for e′ ∈ metaInfo[e] do
36:                COLLECTDFS(e′)
37:    for e ∈ E do
38:        if e's label is the start symbol then
39:            COLLECTDFS(e)
40:    return con
```

The following theorem demonstrates the correctness of Algorithm 2.

Theorem 13 (Tracing Algorithm's Correctness)**.** *For any instance of CFL-reachability* $\langle C, (V, E)\rangle$*, we have*

$$\textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle)) = \mathsf{Ctri}(C, (V, E)).$$

Proof. Suppose $e \in \textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle))$. According to Algorithm 2, we have $e \in E$, and there exists a finite sequence of edges $e_1 = e, e_2, \ldots, e_n$ (we can move e to the beginning), where each edge is either from the original edge set or obtained from applying one production in C to some preceding edges, e directly or indirectly contributes to e_n, and e_n's label is C's start symbol. So $e \in \mathsf{Ctri}(C, (V, E))$. Conversely, suppose $e \in \mathsf{Ctri}(C, (V, E))$, then $e \in E$ and there exists a finite sequence of edges $e_1 = e, e_2, \ldots, e_n$, where each edge is either from the original edge set or obtained by applying one production in C to some preceding edges, e directly or indirectly contributes to e_n, and e_n's label is C's start symbol. According to Theorem 12, every edge in this sequence must be added to the edge set by Algorithm 1 (and thus also by Algorithm 2). Now let us consider e_n. It is not in the original edge set because its symbol is a non-terminal symbol. Thus e_n can only be obtained by applying one production on some previous edges $\{e_{i_1}, e_{i_2}, \ldots, e_{i_{n_i}}\}$ (which are called the dependency edges of e_n). Since all dependency edges of e_n were added by the algorithm, one edge must be the last one added, and when that one was added, all of e_n's dependency edges were added to the meta-information of e_n (*i.e.*, $\mathsf{metaInfo}(e_n)$). Thus the COLLECTDFS procedure can visit all dependency edges of e_n. Such dependency edges can be reasoned similarly in a depth-first-search manner, and because e_1 directly or indirectly contributes to e_n, eventually COLLECTDFS visits the edge $e_1 = e$. So $e \in \textsc{collect}(\textsc{record}(\langle C, (V, E)\rangle))$. □

Complexity Analysis for Algorithm 2. For RECORD, suppose the grammar of the CFL is fixed, the graph supports constant time edge addition and linear time adjacent ver-

tices traversal, the worklist supports constant time pushing/popping, and the operations on metaInfo and the edge set corresponding to each key have logarithmic time complexity and linear space complexity with respect to the number of elements. There could be at most $\tilde{O}(|V|^2)$ edges popped out from the worklist at line Line 10, and for each edge popped out, there could be at most $\tilde{O}(|V|)$ adjacent edges to try in the main **while** loop. The size of metaInfo is bounded by $\tilde{O}(|V|^2)$ and the size of the edge set corresponding to each key is bounded by $\tilde{O}(|V|)$, so each addition operation to metaInfo has a complexity of $\tilde{O}(\log(|V|^2) + \log|V|) = \tilde{O}(\log|V|)$, which can be hidden by our $\tilde{O}$ notation. Thus the time complexity of RECORD is $\tilde{O}(|V|^3)$. The space complexity of RECORD is $\tilde{O}(|V|^3)$ since metaInfo dominates the space complexity and there could be at most $\tilde{O}(|V|^3)$ edge additions to metaInfo. For COLLECT, suppose visited and con also supports logarithm time operations. Consider a new graph where vertices are edges in E, and if $e_2 \in \mathsf{metaInfo}[e_1]$ then we have a "super edge" from e_1 to e_2. It is easy to see that in this new graph, there are at most $\tilde{O}(|V|^2)$ vertices, and the in-degree and out-degree of each vertex are both bounded by $\tilde{O}(|V|)$. Then COLLECT essentially did a depth-first search on this new graph, whose time complexity is determined by the maximum number of "super edges" in this new graph: $\tilde{O}(|V|^2 \times |V| \times c \cdot \log|V|) = \tilde{O}(|V|^3)$. The logarithm factor is due to operations on visited, metaInfo, and con, but is hidden by our notation $\tilde{O}$. The space complexity is bounded by the sizes of the graph ($\tilde{O}(|V|^2)$), visited ($\tilde{O}(|V|^2)$), con ($\tilde{O}(|V|^2)$), metaInfo ($\tilde{O}(|V|^3)$), and the maximum depth of recursive calls ($\tilde{O}(|V|^2)$). Therefore, the space complexity of COLLECT is $\tilde{O}(|V|^3)$.

Table 5.1 compares the time/space complexities of the standard CFL-reachability algorithm (Algorithm 1) and our tracing version (Algorithm 2). In practice, however, the running time and space also depend on the constant and logarithm factors, the computer architecture, etc.

Table 5.1: Time/space complexities of Algorithm 1 and Algorithm 2. The CFL size is assumed to be a constant, and the input graph is $G = (V, E)$.

Algorithm	Time Complexity	Space Complexity
Algorithm 1 (The Standard Algorithm)	$\tilde{O}(\|V\|^3)$	$\tilde{O}(\|V\|^2)$
Algorithm 2 (Our Tracing Algorithm)	$\tilde{O}(\|V\|^3)$	$\tilde{O}(\|V\|^3)$

5.4.4 Mutual Refinement Algorithm

This section precisely defines the mutual refinement algorithm for a computationally hard L-reachability problem with multiple CFL approximations.

Definition 16 (Refinement Sequence)**.** *Given an instance $\langle L, (V, E)\rangle$ of L-reachability problem and m different CFLs $C_1, \ldots, C_m$ ($m \geq 2$), each of which over-approximates L, a refinement sequence is a finite sequence of sets of edges $E_1, \ldots, E_n$, such that $E_1 = E$ and for all $i \geq 2$, E_i is either $\mathsf{Ctri}(C_j, (V, E_k))$ where $j \in \{1, 2, \ldots, m\}$ and $k \in \{1, 2, \ldots, i-1\}$, or $E_j \cap E_k$ where $j, k \in \{1, 2, \ldots, i-1\}$.*

There exists a global minimum edge set (with respect to the set inclusion relation) that can be computed via a fix-point algorithm. This is due to the following monotonicity property of contributing edges.

Lemma 2 (Monotonicity of Contributing Edges)**.** *Given a formal language L, the following formula holds for all problem instances.*

$$E_1 \subseteq E_2 \implies \mathsf{Ctri}(L, (V, E_1)) \subseteq \mathsf{Ctri}(L, (V, E_2))$$

Proof. This is because any L-path in E_1 is also an L-path in E_2. □

Theorem 14 (Minimum Edge Set)**.** *Given an instance $\langle L, (V, E)\rangle$ of the L-reachability problem and m different CFLs $C_1, C_2, \ldots, C_m$ ($m \geq 2$), each of which over-approximates L, there exists an edge set $E_{\min}$, which could be obtained by a specific refinement sequence, and which is a subset of all edge sets in all refinement sequences.*

Proof. Consider the process of applying m algorithms successively in an arbitrary order to refine the edge sets: apply C_{i_1}-reachability on E to get E_1, apply C_{i_2}-reachability on E_1 to get E_2, ..., apply C_{i_m}-reachability on E_{m-1} to get E_m, where $C_{i_1}, C_{i_2}, \ldots, C_{i_m}$ is an arbitrary permutation of $C_1, C_2, \ldots, C_m$. For simplicity, we still denote this order as $C_1, C_2, \ldots, C_m$. This is called one round. By doing such rounds multiple times until a state where applying another round does not change the edge set, we get the set $E_{\min}$.

First, we prove its termination: after each round, the size of E either decreases or remains the same, but the size of E cannot decrease indefinitely.

Second, we show that $E_{\min}$ is indeed the globally minimal set with respect to the subset relation: given any refinement sequence $E_1, \ldots, E_n$, since it is non-increasing with respect to the set inclusion relation, we just need to prove $E_{\min} \subseteq E_n$. We prove this by induction on the lengths of refinement sequences. First, there is only one possible refinement sequence of length 1, which is E itself, and it is obvious that $E_{\min} \subseteq E$. Now suppose that for all refinement sequences of length at most n, $E_{\min}$ is a subset of the last edge set in the sequence. Consider an arbitrary refinement sequence of length $n+1$: $E_1, \ldots, E_{n+1}$. Here E_{n+1} is either the intersection of two previous edge sets or obtained by applying C_j-reachability ($j \in \{1, \ldots, m\}$) to one of the previous edge sets. In the first case, by the induction hypothesis, the two previous edge sets all contain $E_{\min}$, so E_{n+1} also contains $E_{\min}$. In the second case, suppose C_j-reachability is applied to E_i ($1 \leq i \leq n$). By the induction hypothesis, we have $E_{\min} \subseteq E_i$, and because the set of contributing edges is monotonic in the sense described in Lemma Lemma 2, we further have $E_{\min} = \mathsf{Ctri}(C_j, (V, E_{\min})) \subseteq \mathsf{Ctri}(C_j, (V, E_i)) = E_{n+1}$, where the first equality holds according to the definition of $E_{\min}$. □

Algorithm 3 The Mutual Refinement (MR) Algorithm

```
1: function MR((V, E), {C_1, ..., C_m})
2:     G ← (V, E)
3:     while true do
4:         s ← G.E.size()
5:         for C_i ∈ {C_1, ..., C_m} do
6:             G.E ← COLLECT(RECORD(⟨C_i, G⟩))
7:         if G.E.size() == s then
8:             return G
```

Algorithm 3 (mutual refinement) gives the complete procedure for finding the minimum edge set described in Theorem 14. It keeps iterating over the given set of algorithms (line Line 5) until the edge set's size does not change (line Line 7). Due to Theorem 14, this algorithm is guaranteed to terminate and produce the optimal result among all possible refinement sequences.

Theorem 15 (Mutual Refinement Soundness)**.** *If every CFL in* $\{C_1, \ldots, C_m\}$ *over-approximates* L*, then Algorithm 3 does not miss any* L*-contributing edges.*

Proof. This is immediate from Theorem 13 and Theorem 14. □

The final reachability result can be obtained by executing C_1, C_2, …, C_m-reachability on the minimum graph produced by Algorithm 3, and reporting the pairs that all those CFL-reachability executions report as reachable. In fact, the last iteration of the while loop in Algorithm 3 already does this.

Theorem 16 (Precision Guarantee)**.** *Suppose we have an instance* $\langle L, (V, E)\rangle$ *of* L*-reachability problem and* m *different CFLs* $C_1, \ldots, C_m$*, each of which over-approximates* L*. Let* E_{min} *be the set of edges obtained by executing Algorithm 3 on* (V, E) *and* $C_1, \ldots, C_m$*. Suppose* $P_1, \ldots, P_m$ *are the sets of reachable pairs obtained by executing* $C_1, \ldots, C_m$ *reach-*

abilities on (V, E)*, and* $Q_1, \ldots, Q_m$ *are the sets of reachable pairs obtained by executing* $C_1, \ldots, C_m$ *reachabilities on* (V, E_{min})*. Then* $\bigcap_{i=1}^{m} Q_i \subseteq \bigcap_{i=1}^{m} P_i$*.*

Proof. Since $E_{\mathsf{min}} \subseteq E$, it is immediate that $\forall i \in \{1, \ldots, m\}, Q_i \subseteq P_i$, because any C_i-path connecting two vertices in (V, E_{min}) is also present in (V, E). □

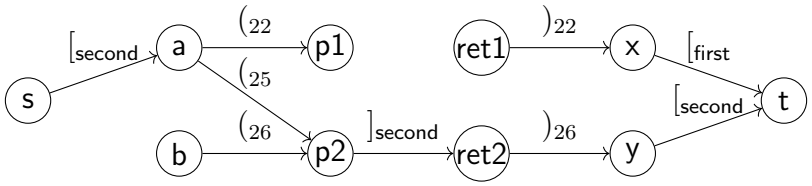
The time and space complexities of Algorithm 3 depend on the number of iterations, which highly depends on the graph structure and the CFLs. $O(|E|)$ is a very loose upper bound of the number of iterations because in each iteration before the last one, we remove at least one edge. Our evaluation (Section 5.5) shows that for specific program analysis problems and graphs with edge sizes up to 184k, the number of iterations can still be within five.

Example 6 (Mutual Refinement Example)**.** *If we apply mutual refinement to the motivating example discussed in Section 5.2, where in each round we apply* C_B*-reachability and* C_P*-reachability in sequence, then after two rounds, the graph stabilizes. Figure 5.6 shows this process.*

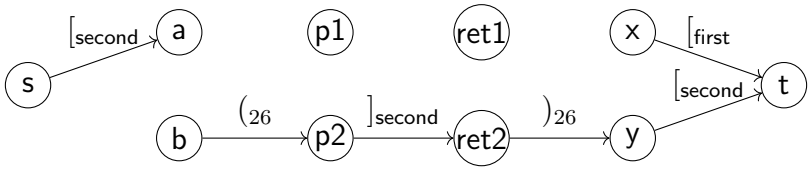
5.5 Experiments

We evaluate mutual refinement on two applications: a taint analysis for Java programs obtained from Android apps [88], and a value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [89].

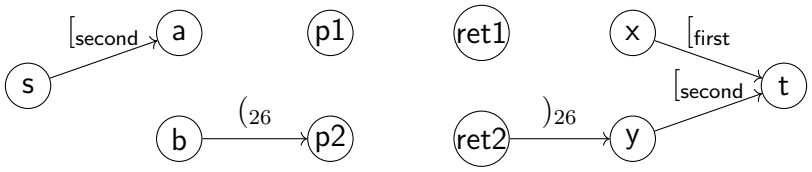
When processing experimental data, we use arithmetic means ($\frac{1}{n}\sum_{i=1}^{n} x_i$) for the average of absolute numbers, and use geometric means ($\sqrt[n]{\prod_{i=1}^{n} x_i}$) for the average of ratios [91]. Also, for the measurement of precision (the number of reachable pairs), we exclude trivial pairs (u, u) and only consider pairs (u, v) where $u \neq v$.



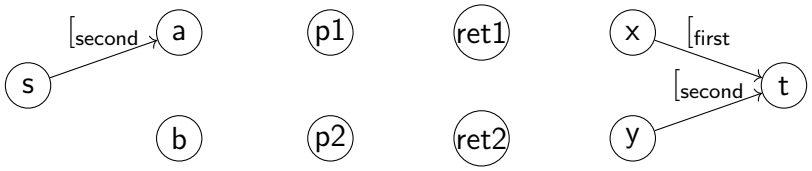
(a) After running C_B-reachability in the first round



(b) After running C_P-reachability in the first round



(c) After running C_B-reachability in the second round



(d) After running C_P-reachability in the second round

Figure 5.6: Mutual refinement's iteration process on the motivating example discussed in Section 5.2. It takes two rounds to converge. If we only consider (s, t)-reachability, then the iteration can stop after the second iteration.

5.5.1 Experimental Setup

Taint Analysis. We apply our approach to a context-sensitive field-sensitive taint analysis for Java programs obtained from Android apps [88]. The analysis goal is to determine all pairs of variables (s, t) where sensitive information from variable s can flow into variable t. Parentheses model context-sensitivity and brackets model field-sensitivity. A valid path string is an arbitrary interleaving of two strings derived from the two CFLs P and B shown

$$
\begin{aligned}
P &\rightarrow P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_k\ P\ )_k \mid \epsilon \\
B &\rightarrow B\ B \mid [_1\ B\ ]_1 \mid \ldots \mid [_l\ B\ ]_l \mid \epsilon \\
L &= P \odot B
\end{aligned}
$$

(a) The taint analysis is formulated as the well-known interleaved-Dyck-reachability problem.

$$
\begin{aligned}
C_P &\rightarrow C_P\ C_P \mid (_1\ C_P\ )_1 \mid \ldots \mid (_k\ C_P\ )_k \mid I \mid \epsilon \\
I &\rightarrow [_1 \mid]_1 \mid \ldots \mid [_l \mid]_l \\
\\
C_B &\rightarrow C_B\ C_B \mid [_1\ C_B\ ]_1 \mid \ldots \mid [_l\ C_B\ ]_l \mid J \mid \epsilon \\
J &\rightarrow (_1 \mid)_1 \mid \ldots \mid (_k \mid)_k \mid \epsilon
\end{aligned}
$$

(b) We use the above two CFLs C_P and C_B to over-approximate the interleaved-Dyck language.

Figure 5.7: The taint analysis formulation and approximation.

in Figure 5.7a. This is the interleaved-Dyck reachability problem, which is undecidable [4].

Unlike the example in Section 5.2, which considers sources/sinks within one function (matched parentheses) and counts flowing into fields as leaks (unmatched brackets), in our experiments, we only count leaks from variables to variables, thus disallowing unmatched brackets. One reason is that this is the formulation in the original work [88], and the other reason is that we also evaluate the LZR algorithm [78] on these benchmarks, which only supports matched brackets. In general, the grammar can be adjusted according to needs.

To use mutual refinement, we choose two CFLs (C_P, C_B) over-approximating the interleaved-Dyck language L, where C_P models parentheses matching and C_B models brackets matching. Their grammars are shown in Figure 5.7b. The execution order is C_P, C_B in each round of mutual refinement.

The benchmarks are selected from the original paper [88]. Specifically, we chose the Contagio malware apps and used the implementation of the client analysis to obtain the graphs.[2] We excluded benchmarks that our tool or the original reference's tool failed to handle. Finally, we got 15 benchmarks, and the size information of the APK files and the

[2]https://github.com/proganalysis/type-inference.

graphs is shown in Table 5.2a.

Value-Flow Analysis. We also apply our approach to a context-sensitive value-flow analysis for LLVM IR programs obtained from the SPEC CPU 2017 benchmark [89]. The analysis goal is to determine all pairs of store/load instructions $(\mathsf{store}\ v_1\ \mathsf{to}\ p_1, \mathsf{load}\ v_2\ \mathsf{from}\ p_2)$ where the value of v_1 can flow into v_2 via intermediate assignments and loads/stores. In this case, context-sensitivity is modeled using matched parentheses; memory stores and loads are modeled using matched brackets (in this case, there is only one type of brackets); normal copies of values are modeled using edges with a special label n. Furthermore, since we are interested in store/load pairs, the first edge in the path string must be a memory store, and the last edge must be a memory load. A valid path string is an interleaving of three strings derived from the three CFLs P, B, and N shown in Figure 5.8a where the first symbol must be $[$ and the last symbol must be $]$. We denote this formal language as L_V. Section 5.6 shows the existing $D_1 \odot D_k$-reachability problem, whose decidability is currently open [92], is reducible to the L_V-reachability problem.

In order to use mutual refinement, we choose three CFLs (C_P, C_B, and C_E) over-approximating the underlying problem, where C_P models parentheses matching, C_B models brackets matching, and C_E enforces that the first and last edges of the path must be an open bracket and a closed bracket, respectively. Specifically, they have the three grammars shown in Figure 5.8b. The execution order is C_P, C_B, C_E in each round of mutual refinement.

The benchmarks are compiled using Clang version 12 [93] to bitcode files. The graphs are generated by the open-source static value-flow analysis framework SVF [94]. We did not include small programs (with bitcode file sizes $<$ 1MB) or programs that failed to be compiled or linked. Finally, we got 10 benchmarks, and the size information of the bitcode files and the graphs is shown in Table 5.2b.

Research Questions. Our experiments aim to answer the following questions.

- *RQ1:* Can mutual refinement achieve better precision compared with the straightfor-

$$
\begin{aligned}
P &\to P\ P \mid (_1\ P\ )_1 \mid \ldots \mid (_k\ P\ )_k \mid \epsilon \\
B &\to B\ B \mid [\ B\ ] \mid \epsilon \\
N &\to n\ N \mid \epsilon \\
L_V &= ((P \odot B) \odot N) \cap \{s \mid s = [*]\}
\end{aligned}
$$

(a) The value-flow analysis is formulated as an L_V-reachability problem.

$$
\begin{aligned}
C_P &\to C_P\ C_P \mid (_1\ C_P\ )_1 \mid \ldots \mid (_k\ C_P\ )_k \mid I \mid \epsilon \\
I &\to [|\ ] \mid n \\
\\
C_B &\to C_B\ C_B \mid [\ C_B\ ] \mid J \mid \epsilon \\
J &\to (_1|\ )_1 \mid \ldots \mid (_k|\ )_k \mid n \\
\\
C_E &\to [\ K\ ] \\
K &\to K\ K \mid [|\ ] \mid (_1|\ )_1 \mid \ldots \mid (_k|\ )_k \mid n
\end{aligned}
$$

(b) We use the above three CFLs C_P, C_B, and C_E to over-approximate L_V.

Figure 5.8: The value-flow analysis formulation and approximation.

ward intersection (baseline) on the two applications?

- *RQ2:* What is the time/space overhead that mutual refinement incurs compared with the straightforward intersection (baseline), and how many rounds does mutual refinement take to converge?

- *RQ3:* Can the LZR graph simplification algorithm improve the precision/performance of mutual refinement on the taint analysis application?

Implementation and Experiment Execution. We implemented mutual refinement in C++17.[3] All experiments were performed on a machine running Ubuntu 20.04.2 LTS. We set a time-out of 4 hours and a space limit of 128 GB for each algorithm's execution on each benchmark item. For *RQ3*, we used the original implementation of the LZR algorithm available

[3]The implementation is available on GitHub (https://github.com/sdingcn/mutual-refinement) and Zenodo (https://doi.org/10.5281/zenodo.8191389). Certain low-level data structure optimizations were used.

Table 5.2: Benchmark statistics.

Benchmark	APK Size (M)	Graph Size ($\|V\|, \|E\|$)
backflash	0.75	(544, 2048)
batterydoc	0.51	(1674, 4790)
droidkongfu	0.08	(734, 1983)
fakebanker	5.17	(434, 1103)
fakedaum	0.14	(1144, 2603)
faketaobao	0.44	(222, 450)
jollyserv	0.42	(488, 998)
loozfon	0.04	(152, 323)
phospy	0.18	(4402, 15660)
roidsec	0.03	(553, 2026)
scipiex	0.31	(1809, 5820)
simhosy	1.43	(4253, 13768)
skullkey	6.63	(18862, 69599)
uranai	0.07	(568, 1246)
zertsecurity	0.10	(281, 710)

(a) Taint Analysis Graphs.

Benchmark	Bitcode Size (M)	Graph Size ($\|V\|, \|E\|$)
cactus	5.61	(101325, 114805)
imagick	13.68	(103594, 131707)
leela	2.93	(16134, 19110)
nab	1.41	(12727, 13605)
omnetpp	20.80	(171502, 184601)
parest	16.20	(84355, 93493)
perlbench	11.88	(125345, 160958)
povray	7.38	(61802, 71892)
x264	4.68	(49806, 56376)
xz	1.24	(9918, 10767)

(b) Value-flow Analysis Graphs.

online.[4] Since the LZR algorithm is fast enough, we did not set time/space limits on its executions.

5.5.2 RQ1: Precision Improvement

According to Theorem 16, mutual refinement's precision is at least as good as the straightforward intersection. We define the precision improvement as $(P_{\mathsf{Baseline}}/P_{\mathsf{MR}}) - 1$, where P_{Baseline} and P_{MR} represent the number of reachable pairs computed by the straightforward intersection (baseline) and mutual refinement, respectively. Table 5.3 shows that, on average, mutual refinement achieves 50.95% precision improvement on the taint analysis benchmarks and 9.37% precision improvement on the value-flow analysis benchmarks. Note that the improvement greatly depends on specific applications and benchmarks.

Summary: On the two program analysis applications, mutual refinement can achieve visibly better precision compared with the straightforward intersection.

[4] https://github.com/yuanboli233/interdyck_graph_reduce.

5.5.3 RQ2: Performance Overhead

Mutual refinement traces the sets of contributing edges, which can cost more time and space. Also, mutual refinement might need several rounds to converge. As shown in Table 5.3 and Figure 5.9, on average, on the taint analysis benchmarks, mutual refinement takes 2.93 rounds to converge and consumes $2.65\times$ time and $3.23\times$ memory compared with the baseline; on the value-flow analysis benchmarks, mutual refinement takes 3.13 rounds to converge and consumes $2.55\times$ time and $2.22\times$ memory compared with the baseline. In some cases, mutual refinement's space consumption can be much higher. For example, mutual refinement incurs a $80.58\times$ space increase on the phospy benchmark in Table 5.3a. And there is a trend that larger graphs result in larger differences in memory consumption, which reflects the space complexity difference ($\tilde{O}(|V|^2)$ and $\tilde{O}(|V|^3)$). Section 5.6.4 discusses mutual refinement's memory cost in detail.

However, mutual refinement can also simplify the graph during the execution of each CFL-reachability, while the straightforward intersection cannot. This could lead mutual refinement to consume less resources in certain cases. For example, on the cactus benchmark in Table 5.3b, mutual refinement consumes less time than the straightforward intersection.

Summary: Mutual refinement typically needs more time and space, but the average time/space increase on the two program analysis applications is within $5\times$, and the number of iterations needed to converge is within five.

5.5.4 RQ3: Combination with the LZR Algorithm

The LZR graph simplification algorithm [78] works for interleaved-Dyck-reachability, and is thus applicable to our taint analysis benchmarks as a pre-processing step. One important detail is that the LZR algorithm does graph edge contractions before calculating the contributing edges, while mutual refinement doesn't. This can lead to edges counted as contributing edges in mutual refinement but not counted as contributing edges in the LZR algorithm, because contracting edges can make LZR ignore the contracted edges. As a

Table 5.3: Precision and performance results. We present the number of rounds that mutual refinement takes to converge, as well as the comparison of precision/time/space between the straightforward intersection (baseline) and mutual refinement. "-" means time/space limits are exceeded.

Benchmark	Iterations	Precision (Pairs)		Time (Seconds)		Space (MB)	
		Baseline	MR	Baseline	MR	Baseline	MR
backflash	2	6080	2870	0.42	0.67	16.06	36.70
batterydoc	3	8386	5484	0.93	6.06	31.43	133.76
droidkongfu	3	6471	5442	0.32	4.55	16.93	90.06
fakebanker	3	1407	1172	0.04	0.12	9.17	11.96
fakedaum	3	3507	3243	0.28	1.07	18.16	37.40
faketaobao	3	398	328	0.01	0.02	6.98	7.23
jollyserv	3	562	303	0.10	0.14	10.61	15.62
loozfon	2	441	424	0.01	0.04	6.74	9.62
phospy	3	103961	81925	1309.64	9436.26	1202.29	96882.00
roidsec	2	17301	16425	1.79	5.94	28.85	168.93
scipiex	4	20542	10210	69.96	266.33	209.97	3471.78
simhosy	3	100552	41992	102.10	110.78	363.91	1669.46
skullkey	-	-	-	-	-	-	-
uranai	4	353	148	0.11	0.08	10.54	10.56
zertsecurity	3	1969	1110	0.26	0.16	11.24	13.93

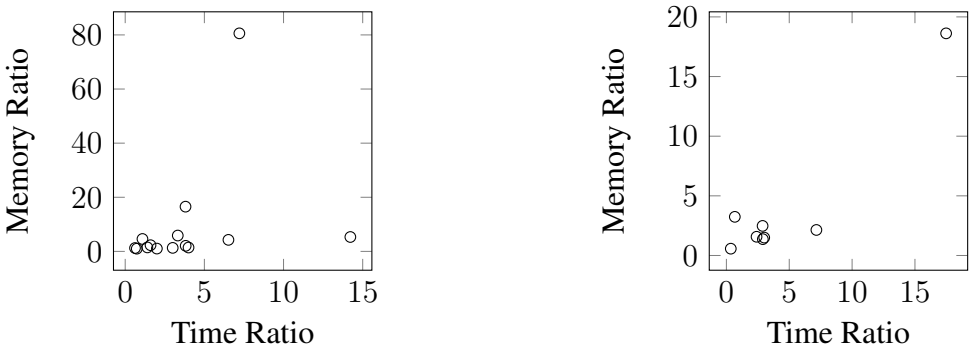
(a) Taint Analysis Results.

Benchmark	Iterations	Precision (Pairs)		Time (Seconds)		Space (MB)	
		Baseline	MR	Baseline	MR	Baseline	MR
cactus	4	46502	46421	621.36	419.39	2888.79	9373.77
imagick	-	22091	-	14088.50	-	12211.74	-
leela	3	392	392	0.81	1.94	78.01	122.45
nab	3	1958	1788	0.30	0.87	51.83	70.76
omnetpp	4	90412	50568	76.70	221.21	1769.57	4396.33
parest	3	4571	4571	2.89	8.70	243.79	364.86
perlbench	-	-	-	-	-	-	-
povray	3	7453	7230	18.18	6.43	455.19	260.73
x264	3	61577	60792	47.01	821.81	571.96	10650.62
xz	2	211	211	0.32	2.29	41.10	87.94

(b) Value-flow Analysis Results.

result, the LZR algorithm can potentially remove certain edges that mutual refinement cannot.

Table 5.4 compares (1) executing mutual refinement on the original graphs and (2) ex-



(a) Overhead on taint analysis.

(b) Overhead on value-flow analysis.

Figure 5.9: Mutual refinement's performance overhead scatter plots (ratios). Time ratios are mutual refinement's time consumption numbers divided by the baseline's time consumption numbers. Memory ratios are similar.

ecuting mutual refinement on the graphs simplified by the LZR algorithm. In the (2) case, the time/space consumption includes both algorithms (LZR and MR). On average, LZR can improve the precision of mutual refinement by 3.27%, and this relatively small number shows that mutual refinement itself can already achieve very high precision. Indeed, LZR+MR can reduce 81.38% more edges on average compared with LZR alone. LZR can also boost mutual refinement in terms of time/space consumption, such as the phospy benchmark in Table 5.4. Notably, the space consumption of LZR+MR is always lower-bounded by 269 MB, and that is because LZR has a minimal memory consumption of roughly 269 MB. This might be due to implementation details.

Summary: LZR can improve mutual refinement's precision to a small extent (3.27% on average). Since LZR is fast, it can boost mutual refinement's performance in certain cases.

Table 5.4: A comparison between the original mutual refinement and the one combined with the LZR algorithm, including precision, time, space, and edge reduction. "-" means time/space limits for mutual refinement are exceeded.

Benchmark	Precision (Pairs)		Time (Seconds)		Space (MB)		Edge Reduction	
	MR	LZR+MR	MR	LZR+MR	MR	LZR+MR	LZR	LZR+MR
backflash	2870	2870	0.67	0.60	36.70	269.18	677	1434
batterydoc	5484	5438	6.06	1.54	133.76	269.33	1826	3327
droidkongfu	5442	5422	4.55	2.66	90.06	269.40	589	1070
fakebanker	1172	928	0.12	0.17	11.96	269.17	414	745
fakedaum	3243	3243	1.07	0.87	37.40	269.18	1103	1747
faketaobao	328	325	0.02	0.10	7.23	269.13	191	309
jollyserv	303	303	0.14	0.25	15.62	269.01	361	634
loozfon	424	424	0.04	0.08	9.62	269.08	105	189
phospy	81925	80200	9436.26	8849.91	96882.00	69532.63	3838	6659
roidsec	16425	16425	5.94	5.86	168.93	269.12	605	1077
scipiex	10210	10173	266.33	260.05	3471.78	3426.72	1219	2683
simhosy	41992	35419	110.78	78.66	1669.46	1217.93	4801	8866
skullkey	-	-	-	-	-	-	19408	-
uranai	148	148	0.08	0.21	10.56	269.10	566	1063
zertsecurity	1110	1110	0.16	0.22	13.93	269.11	253	438

5.6 Discussion

5.6.1 Generality of Mutual Refinement

Mutual refinement can approximate any language-reachability problem as long as there exist CFL-reachability-based over-approximations. In particular, it is not restricted to the interleaved-Dyck reachability or any particular problem. We have shown two examples L_T and L_V in our motivating example and experiments, and here we show (1) L_T is not a CFL, and (2) $D_1 \odot D_k$-reachability, whose decidability is currently open [92], is reducible to L_V-reachability.

L_T is not a CFL. Suppose L_T is a CFL. We construct a formal language M, each string m of which is an arbitrary interleaving of $p \in P$ and $d \in D$ in Section 5.2, interspersed with an arbitrary number of special symbols $a_1, \ldots, a_l$. Obviously, the language homomorphism

$$f(x) = \begin{cases} [_i & , x = a_i \\ x & , \text{otherwise} \end{cases}$$

maps M to L_T. By the assumption L_T is context-free, so M is also context-free since CFL is closed under inverse homomorphisms. Consider the regular language $R = \{s \mid s \text{ does not contain } a_1, \ldots, a_l\}$. Since the intersection of a context-free language and a regular language is also context-free, we have $M \cap R$ is context-free, but this is the interleaved-Dyck language, which is known to be non-context-free [4]. This is a contradiction.

$D_1 \odot D_k$-reachability is reducible to L_V-reachability. $D_1 \odot D_k$ is arbitrary interleaving of two Dyck languages D_1 and D_k, where D_1 is a Dyck language with one kind of parenthesis and D_k is a Dyck language with k kinds of parenthesis. Given any labeled graph consisting of only labels from D_1 and D_k, a pair of vertices (s, t) is $D_1 \odot D_k$-reachable if and only if (s_0, t_0) is L_T-reachable where we just add two vertices s_0 and t_0, and two edges $s_0 \xrightarrow{[} s$ and $t \xrightarrow{]} t_0$, where [and] are the open and close brackets in D_1.

5.6.2 Different Grammars for the Same CFL

A context-free language can be represented by many different context-free grammars. Ambiguous grammars introduce redundancies, so it can affect mutual refinement's performance because there are more derivations to traverse. However, the choice of grammars does not affect the precision, because we only track L-contributing edges and different grammars refer to the same formal language L. This is also reflected in Algorithm 2: $\mathsf{metaInfo}[e]$ is a set eliminating duplicate tracked edges, and con is also a set eliminating duplicate collected edges.

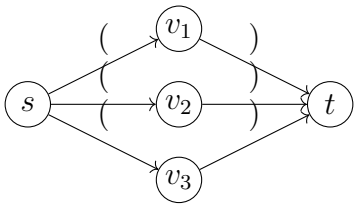
5.6.3 Order of Mutual Refinement

In mutual refinement, as Theorem 14 shows, any possible orders of executing the CFL-reachability-based over-approximations C_1, C_2, $\cdots$, C_m result in the same global minimum. However, different orders might affect the convergence speed. In practice, we can run the available CFL-reachability-based over-approximations on sampled programs to find out a "good" order to use, and then execute the "good" order on all programs. There are other heuristics for order choosing, such as executing the one that can result in the best precision first.

5.6.4 Cost of Mutual Refinement

As shown in Section 5.5, mutual refinement, in general, needs more time and space than the straightforward intersection. This is because mutual refinement traces the contributing edges and might need more than one iteration to converge. However, after running one CFL-reachability over-approximation, the remaining ones only need to be executed on the simplified graph. Also, in practice, we do not have to wait for the convergence, but can run it for a fixed number of rounds (e.g. two rounds). Other possible optimizations include changing the order of mutual refinement, simplifying the graphs/grammars, etc. Mutual refinement reflects a trade-off between performance and precision.

Memory Overhead of Mutual Refinement. Our experiment shows that in some cases, mutual refinement can take about $80\times$ memory compared with the straightforward intersection. We intuitively explain the reason. Consider the difference between the standard CFL-reachability algorithm (Algorithm 1) and our tracing version (Algorithm 2): in our tracing version, when a new edge is generated, the meta-information about edge dependencies is updated no matter whether the new edge is already in the graph or not. If there are multiple ways to generate the same edge, all of those ways need to be recorded. For example, in the following graph, where we perform the matched-parenthesis reachability with respect to the grammar $S \to S\ S \mid (\ S\ ) \mid \epsilon$, there are three ways to generate the summary edge $s \xrightarrow{S} t$, and all edges will be added to the meta information of $s \xrightarrow{S} t$, despite there is only one such summary edge in the final graph. So the memory cost of mutual refinement can be high. Whether this graph pattern occurs in reality depends on the specific analysis details.



Exploring whether we can reduce the memory cost is an interesting future direction. There exists work compressing information used during static analysis [95].

5.6.5 Generalization to the Single-Pair Case

In this chapter, mutual refinement is formalized to retain the edges contributing to reachable pairs in the CFL-reachability-based over-approximations. Notice that in the single-source-single-target reachability case, we can retain only the edges contributing to the pair that we are interested in, and this can potentially remove even more edges from the graph and thus can also potentially increase the precision as well. We leave this for future work.

5.6.6 Generalization to Other Algorithms

The idea of tracing in mutual refinement can be potentially generalized to all algorithms using similar "dynamic programming style" approaches. Specifically, as long as the algorithm traverses all edges contributing to the ground truth solution, we can use tracing to extract those edges and use this as a refinement between different such algorithms. It is an interesting future direction to explore the generalization of mutual refinement to broader classes of algorithms. In particular, the computable witness functions described in Chapter 3 imply that further precision improvements are always "computably" available.

5.7 Related Work

CFL-reachability is widely-used in program analysis [64, 65, 66, 67, 68, 69]. It has a (sub)cubic-time dynamic programming style algorithm [71, 70, 72, 73], and faster algorithms exist in special cases [74, 75, 76]. CFL-reachability can model function calls/returns [66, 4], field reads/writes [79, 80], locks/unlocks [81], etc.

In static analysis, many techniques have been proposed to reduce the size of graphs involved in the analysis [78, 96, 97]. Our mutual refinement process simplifies the graphs, but our main focus is leveraging the information of each CFL-reachability-based over-approximation to refine the results. In particular, the LZR fast graph simplification work [78] also defines similar concepts such as contributing edges, but their algorithm is specific for the interleaved-Dyck-reachability problem, while our mutual refinement works for any L-reachability problems preserving CFL-reachability-based over-approximations. Also, LZR is a pre-processor while mutual refinement is a complete solver.

Interleaved-Dyck-reachability is widely used in program analysis [77, 78, 83], but it is undecidable [4], so there exist many approximation algorithms. We can use one Dyck language to approximate it and employ the standard cubic-time context-free language reachability algorithm [71, 70, 72]. The refinement-based context-sensitive points-to analysis

work [79] used the method of modeling one Dyck language precisely while approximating the other Dyck language using a regular language [79]. The linear conjunctive language reachability [77] is another formulation of interleaved-Dyck-reachability which is precise, but the corresponding algorithm is approximate. Synchronized pushdown systems [83] model the idea of considering two context-free languages at the same time. Mutual refinement is not restricted to interleaved-Dyck-reachability.

In static analysis and verification, similar strategies of running different approaches in a staged way such that later stages benefit from earlier stages have been studied, such as the $\mathsf{Unity} - \mathsf{Relay}$ approach [98] to accumulate the precision of different selective context-sensitivity approaches, and the staged verification [99] where faster verifiers run first to reduce the load of later verifiers. Mutual refinement concerns graph-reachability-based program analysis, and we have theorems showing the existence and uniqueness of fix-points.

5.8 Chapter Conclusion

This chapter proposed mutual refinement to combine different CFL-reachability over-approximations for computationally hard graph reachability problems. We proved theorems showing the existence and uniqueness of the optimal refinement result, the correctness of mutual refinement, and the precision guarantees. To realize mutual refinement, the modifications to the standard CFL-reachability algorithm are minimal, and the modified version's time/space complexities were carefully analyzed. We also conducted experiments showing that mutual refinement achieved better precision than the straightforward intersection of the sets of reachable vertex pairs, with reasonable extra time and space cost.

CHAPTER 6

EXAMPLE: FAST CONSTRAINT SYNTHESIS FOR C++ FUNCTION TEMPLATES

6.1 Introduction

C++ is a high-performance programming language widely used in system programming [100], games and GUI [101], compilers [93], artificial intelligence [102], etc. C++ templates are a powerful language feature that facilitates generic programming and compile-time computations, and this feature has been used extensively in practice. It provides compile-time polymorphism, complementing the run-time polymorphism of virtual functions used in many object-oriented languages. For example, almost all containers and algorithms in the Standard Template Library (STL) utilize templates [103]. Template parameters can be values, types, and templates themselves. It is worth noting that templates can be used to simulate arbitrary Turing machines at compile time [19].

During the compilation process, templates are *instantiated* to generate non-templated C++ code: the compiler substitutes formal template parameters with concrete template arguments. If the concrete template arguments do not support certain operations used in the template bodies, the instantiation fails, and the compiler emits error messages. Because such failures can occur during deeply nested instantiation processes,[1] it is folklore that C++ template errors could be verbose and difficult to understand [105], and this issue has a long history: for example, for a small 26-byte C++ program, g++-4.6.3 can produce 15,786 bytes of output, with the longest line of 330 characters [106]. On the other hand, the diagnostics often involve unnecessary implementation details and do not provide much insight into fixing the errors, which confuses C++ developers and dramatically hinders produc-

[1] The SFINAE mechanism [104] can remove templates from the overload resolution candidate set and thus avoid some errors, but it is often hard to read and maintain.

```
#include <vector>

template <typename T> void f(T x) { std::vector<T> v(x, x); }
int main() { f(nullptr); }

/* abbreviated error message (with manually added "...")
old.cc:3:52: error: no matching constructor for initialization of...
template <typename T> void f(T x) { std::vector<T> v(x, x); }
                                                   ^ ~~~~
old.cc:4:14: note: in instantiation of function template special...
int main() { f(nullptr); }
             ^
.../c++/v1/vector:395:57: note: candidate constructor not viable...
(48 more lines of "candidate constructor not viable" or similar) */
```

(a) Erroneous template instantiation without constraints. Apple Clang 15.0.0 with `-std=c++20` prints 55 lines of error messages.

```
#include <vector>
#include <concepts>

template <std::integral T> void f(T x) { std::vector<T> v(x, x); }
int main() { f(nullptr); }

/* abbreviated error message (with some manually added "...")
new.cc:5:14: error: no matching function for call to 'f'
int main() { f(nullptr); }
             ^
...note: candidate template ignored: constraints not satisfied...
template <std::integral T> void f(T x) { std::vector<T> v(x, x); }
(8 more lines) */
```

(b) Erroneous template instantiation with constraints. Apple Clang 15.0.0 with `-std=c++20` prints 13 lines of error messages.

Figure 6.1: Erroneous C++ template instantiations without/with constraints.

tion [107, 108]. Consider a simple C++ example in Figure 6.1a. The `std::vector` class does not have a constructor suitable for the arguments `(nullptr, nullptr)`. However, C++ compilers are unable to catch this error until the actual instantiation of `std::vector`, resulting in the production of 55 lines of error messages; yet the error messages contain too many implementation details such as "`candidate constructor not viable`" and thus

```
template <typename T>
concept IntClass = requires (T x, T y) {
  typename T::integer_type;
  {x + y} -> std::same_as<T>;
  x.dump();
};

template <typename T> requires IntClass<T> || std::integral<T>
void g(T x) { /* omitted */ }
```

Figure 6.2: Complicated constraints with type requirements, compound requirements, simple requirements, and disjunctions of requirement expressions.

hinder readability.

To improve the readability and maintainability of C++ templates, C++20 introduced a language feature called *constraints and concepts* [109]. Constraints are predicates that impose requirements on template parameters, while concepts are named sets of such requirements. They define clearer interfaces for templates and enable C++ compilers to detect errors early on in the instantiation process (with better error messages). Consider the example in Figure 6.1b, which extends the example in Figure 6.1a by adding the concept `std::integral`. The concept `std::integral`, which is part of the standard library, requires that the template parameters must be of integral types. By using `std::integral`, the error in Figure 6.1b can be caught before the instantiation of `f`'s body, resulting in only 13 lines of error messages. Furthermore, the message "`constraints not satisfied`" is easily understandable. Note that constraints and concepts are very expressive compared to ordinary type systems. For example, the concept `IntClass` in Figure 6.2 specifies three requirements on type `T`: (1) it must contain a type member called `integer_type`; (2) it must support the operator `+` with a result type `T`, and (3) it must support the `dump()` member function call. Additionally, `g`'s template parameter `T` must satisfy either the `IntClass` constraint or the `std::integral` constraint. Thus, manually writing and reasoning about constraints and concepts can be error-prone and time-consuming. It becomes more challenging during the development process with frequent code changes. Moreover, many existing C++ projects do not incorporate concepts or constraints, because these language features were

only introduced in C++20.

The topic of synthesizing constraints and concepts for C++ templates has not been extensively explored. This chapter introduces the first approach to automatically synthesize constraints for C++ function templates based on their template bodies and caller-callee relations. Our approach is built on Clang's frontend and leverages only a lightweight static analysis, thus it is very efficient. Moreover, our approach is fully automated and can be directly applied to real C++ code without requiring manual annotations. The synthesized constraints can both specify clearer interface requirements and significantly improve template-related error messages.

Constraint synthesis for C++ function templates is challenging. Because function templates can call other functions or function templates, the synthesis must be inter-procedural to achieve reasonable precision. However, the caller-callee relation can involve arbitrary argument passing and type correspondence. Moreover, C++ supports function overloading, so the actual function being called inside a function template may not be resolved until instantiations. Recursive dependencies, if present, pose yet another challenge. The Turing-completeness of templates [19] and usages of standard libraries such as <type_traits> eventually made the constraint synthesis problem undecidable (but also opened the door to apply our witness functions described in Chapter 3). Our approach tries to approximate the problem and handles these challenges elegantly and efficiently. We introduce a novel idea called *backmap* to relate type correspondence between the caller and the callee (Section 6.3.3), use disjunctive clauses to model function overloading (Section 6.3.3), and cut off recursive dependencies by treating them as trivial constraints (Section 6.3.4). Finally, we design a polynomial-time simplification procedure for the synthesized constraint formulas (Section 6.3.5).

At first glance, constraints synthesis for C++ templates bears resemblance to type inference [110, 111, 112]. However, there are several key differences. First, instead of inferring an existing type or typeclass, our approach infers a constraint that corresponds

to a set of types, including potential new types that may be defined in the future. Second, the constraints we consider are not limited to those defined by the standard library. In fact, the constraints can incorporate arbitrary conjunctive/disjunctive combinations of member function requirements, member type requirements, and so on. For example, it is valid to define a constraint requiring the member function `specialFunctionA()`, even if no class in the existing `C++` code contains such a member function. Third, as shown in Section 6.2, the precise requirement of a type can be non-computable due to the potential for a non-terminating `C++` compilation if the compiler does not set limits on template instantiations. Therefore, template-related automated reasoning can be viewed as a form of "meta-analysis" of the compilation process.

We have implemented our approach based on the Clang `C++` frontend, targeting function templates, supporting constraints in various forms, including unary operators, binary operators, higher-order functions, class member accesses, and simple type traits. We evaluated our tool on real-world library code from the Standard Template Library (STL) header `<algorithm>` and the Boost library[2] header `<boost/math/special_functions.hpp>`. The evaluation results demonstrate that our tool is efficient and effective. Firstly, our analysis is extremely fast, taking less than 1.5 seconds to process over 110k lines of code (LOC). We are able to synthesize non-trivial constraints for 38.4% of function templates from `algorithm` and for 35.9% of function templates from `special_functions`. Secondly, our analysis is reasonably precise. We select the 14 representative function templates from `algorithm`, as listed in the introductory `C++` textbook [113]. We compare the synthesized constraints with the standard requirements specified in the document. The majority of the synthesized constraints either match or under-approximate the standard requirements. Finally, the synthesized constraints significantly reduce the length of compiler error messages for incorrect instantiations of function templates, with average reductions of 56.6% for `algorithm` and 63.8% for and `special_functions`.

[2]https://www.boost.org.

This chapter makes the following contributions.

- We study the problem of synthesizing template constraints for improving C++ code readability and maintainability.
- We present the first automated constraint synthesis for C++ templates.
- We conduct an extensive evaluation based on two widely used C++ libraries. The empirical results demonstrate that our approach is fast, precise, and can significantly reduce template-based compiler errors.
- We designed and implemented a way to automatically measure the effectiveness of compilation error message reductions.

The rest of the paper is structured as follows. Section 6.2 describes the C++ background and the problem that we target, including its general undecidability. Section 6.3 describes our approach in detail. Section 6.4 gives the experimental results. Section 6.5 contains two case studies for error messages. Section 6.6 discusses more about the spirit of our work and technical details. Section 6.7 surveys related work, and Section 6.8 concludes.

6.2 Preliminary

This section reviews the background and introduces the constraint synthesis problem.

6.2.1 C++ function templates, constraints, and concepts

We use an example to illustrate C++ templates and constraints/concepts. The most common syntax of C++ function template definition consists of a template parameter list followed by the function body, as shown in Figure 6.3a. In this example, the template parameters `T`, `U`, and `V` are unconstrained. Specifically, the keyword **`typename`** only indicates that these template parameters should be "types." However, it is straightforward to see that not all types can be used:

```
template <typename T>
void f(T x) {
    for (int i = 0; i < 3; i++)
        x.dump();
}

template <typename U, typename V>
void g(U x) {
    x.print(100);
    for (int i = 0; i < 10; i++)
        x.print(i);
    V y;
    f(y);
}
```

(a) An example of unconstrained C++ function templates.

```
template <typename T>
concept Dumpable =
    requires (T x) { x.dump(); }

template <Dumpable T>
void f(T x) {
    for (int i = 0; i < 3; i++)
        x.dump();
}

template <typename U, typename V>
requires Dumpable<V> &&
 (requires (U x, int y) {x.print(y);})
void g(U x) {
    x.print(100);
    for (int i = 0; i < 10; i++)
        x.print(i);
    V y;
    f(y);
}
```

(b) The syntax of constraints and concepts in C++20.

Figure 6.3: The syntax of C++ templates, constraints, and concepts.

- a variable of type `T` must support the member function call `dump()`;
- a variable of type `U` must support the member function call `print(int)`;
- type `V` must satisfy the same constraint as `T` does.

To express these constraints, we can either define a standalone concept (`Dumpable`) and replace **`typename`** with it, or use the **`requires`** clause to specify the constraints in-place, as shown in Figure 6.3b. Here the concept `Dumpable` is a named predicate checking whether a variable of the given type supports the `dump()` member function call, and **`requires`** is used both to associate constraints (either pre-defined concepts or directly written constraint expressions) to function templates and to start "requires expressions" that can be used as parts of larger constraints.

6.2.2 Problem statement and undecidability

From Figure 6.3, we can see that the process of writing constraints involves inter-procedural reasoning. For example, in Figure 6.3b, the requirement `Dumpable` originates from `f` and is propagated into `g`, because `g` calls `f`. Real-world `C++` code consists of much more complicated function templates and call graphs (with possibly overloaded callees), and during the development process, frequent code changes make the situation harder. Thus, manually completing the above process is non-trivial and time-consuming. This chapter proposes automated `C++` template constraint synthesis to help the development process. We focus on type parameters of templates because that is the most frequently used feature in generic programming like STL.

Ideally, our goal is to achieve a 100% precise set of requirements for each type parameter of templates. However, because `C++` template is a Turing-complete language, we can demonstrate that precise constraint synthesis is undecidable by slightly modifying the construction proposed by Veldhuizen [19]. Specifically, given an arbitrary closed Turing machine M, we define a **`struct S`** containing a member type `halted_state`, so that in a specific instantiation of `S`, the `halted_state` is

- **`int`** if and only if M halts in the accept state,
- **`double`** if and only if M halts in the reject state,
- undefined if M does not halt at all.

It is well known that there does not exist a total program distinguishing the first two cases even if we allow the program to output arbitrary results for the third case [114]. We can further specify constraints on any type template parameter

```
static_assert(std::is_same<S<...>::halted_state, T>::value);
```

so in general precise constraint synthesis for type template parameters is also non-computable, and practical synthesizers must sacrifice precision in order to guarantee termination. Our

work synthesizes *under-approximations* of the precise requirements. An under-approximation of requirements means specifying fewer requirements, which corresponds to allowing more types. To sum up, we give our problem definition as follows.

> Given a C++ translation unit, for each function template in the translation unit, for each type template parameter of the function template, compute a constraint $\mathcal{C}$ under-approximating the real requirement $\mathcal{R}$ on this parameter. $\mathcal{C}$ is specified using the C++20 constraints and concepts syntax, and $\mathcal{R}$ is the requirements on the parameter implicit in the template body. Suppose the set of types allowed by $\mathcal{C}$ and $\mathcal{R}$ are $S(\mathcal{C})$ and $S(\mathcal{R})$, respectively. An under-approximating constraint allows more types, so we can express our goal as $S(\mathcal{C}) \supseteq S(\mathcal{R})$.

Our under-approximation algorithm proposed in Section 6.3 guarantees termination. On the other hand, we have shown above that there exists a many-one reduction from the halting problem to the decision problem version of constraint synthesis. According to Chapter 3, there exists a computable witness function converting our algorithm to a C++ program on which it is imprecise. Thus, although the algorithm proposed in Section 6.3 does not directly involve any refinement process, the algorithm itself can be the input to the witness function, permitting more precise algorithms to be developed based on the counterexamples.

6.3 Approach

This section formalizes our approach in detail. To aid our discussion, Section 6.3.1 introduces a simplified calculus for modelling C++ function templates; Section 6.3.2 introduces *constraint formulas* which are finally inserted into the source code to restrict type template parameters. Our main approach consists of the following three steps, which is summarized in Figure 6.4.

1. Section 6.3.3 describes the process of scanning template bodies and constructing the

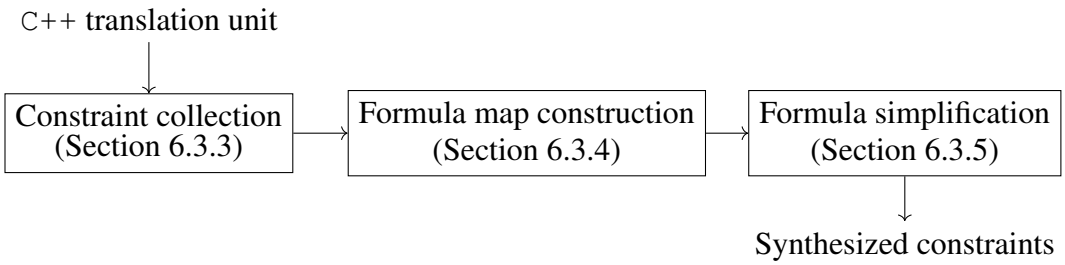


Figure 6.4: An overview of our approach. First, constraint collection (Section 6.3.3) traverses each function template to collect constraints into the inter-procedural constraint map. Second, formula map construction (Section 6.3.4) takes the inter-procedural constraint map and produces the formula graph, which is a compact representation of all constraints and their dependencies in the entire translation unit. Finally, formula simplification (Section 6.3.5) uses a lightweight algorithm to simplify the constraints into versions that are suitable to be inserted into the source code.

inter-procedural constraint map for each translation unit.

2. Section 6.3.4 transforms the inter-procedural constraint map to the *formula graph*, during which recursive dependencies are resolved.

3. Section 6.3.5 describes the simplification process after directly reading the constraint formulas from the formula graph.

Section 6.3.6 discusses the soundness aspect of our approach.

6.3.1 A simplified calculus

C++ is notorious for its complex syntax and semantics, as evidenced by the 1853-page C++20 standard [115]. To address this, we present a simplified calculus (Figure 6.5) that models the core semantics of C++ function templates. This is similar to the spirit of [116], but we also choose to adhere more to the realistic syntax of C++ since our approach applies to real C++ code. This simplified calculus enables us to formally discuss our approach in Section 6.3.2-Section 6.3.5 and discuss soundness properties in Section 6.3.6.

- Because our analysis is flow-insensitive, we omit control flows and only consider the usage sites "use" of the type template parameter t or variables v of the type template

$$
\begin{array}{rcl}
t & \in & \mathsf{Type} \cup \mathsf{TypeParameter} \\
v & \in & \mathsf{Variable} \\
n & \in & \mathsf{FunctionOrFieldName} \\
e & \in & \mathsf{Expression} \\
\mathsf{op_p} & \in & \mathsf{PrefixOperator} \\
\mathsf{op_s} & \in & \mathsf{SuffixOperator} \\
\mathsf{op_i} & \in & \mathsf{InfixOperator} \\
\mathsf{trait} & \in & \mathsf{TypeTrait} \\
\mathsf{use} & := & \mathsf{op_p}\ v \mid v\ \mathsf{op_s} \mid v\ \mathsf{op_i}\ e \mid e\ \mathsf{op_i}\ v \mid v(e^*) \mid v.n \mid v.n(e^*) \mid \mathsf{trait}(t) \mid n(\ldots v \ldots) \\
\mathsf{fun} & := & t\ n((t\ v)^*)\ \{\mathsf{use}^*\} \\
\mathsf{temp} & := & <T^*>\ \mathsf{fun} \\
\mathsf{spec} & := & <>\ \mathsf{fun} \\
\mathsf{translationUnit} & := & (\mathsf{fun} \mid \mathsf{temp} \mid \mathsf{spec})^+
\end{array}
$$

Figure 6.5: A simplified calculus for modelling C++ function templates.

parameter t. The usage sites include direct trait assertions on the type template parameter ($\mathsf{trait}(t)$), and usages of variables of the type template parameter, such as being used as operands ($\mathsf{op_p}v$), functions ($v(e^*)$), member accesses ($v.n$), and arguments ($n(\ldots v \ldots)$, where v should satisfy n's corresponding type requirements).

- A function body "fun" could be non-template functions where the types t involved are all concrete types or template functions where the types t could be type template parameters.

- A function template "temp" consists of a list of type template parameters and a function body. A function template can have specializations "spec" where all type template parameters are substituted by concrete types.

- All of "fun", "temp", and "spec" participate in overloading resolution of function calls.

- A "translationUnit" is a basic compilation unit for this calculus. Our analysis is performed on individual translation units.

6.3.2 Constraint formalization

C++20 supports many kinds of constraints, such as type constraints (*e.g.* requiring the existence of a type member), expression constraints (requiring an expression such as `a.f(1, true)` to successfully compile), type traits (*e.g.* `std::is_same`). These are considered as *atomic constraints*. Our work also supports conjunctions and disjunctions of smaller constraints according to C++20. Formally, we define *constraint formulas* as follows.

Definition 17 (Constraint formula)**.** *Constraint formulas express constraints on types. Atomic formulas pose restrictions on the type template parameter T, and compound formulas are either atomic formulas or conjunctions/disjunctions of smaller formulas. The atomic formulas are similar to the* use *part of our simplified calculus in Figure 6.5, except that we only preserve the type information (e.g., the exact expression e in $v\ \mathsf{op}_i\ e$ is omitted; only its type is preserved). Note that a specific atomic formula* convertible_to *is introduced, which helps to handle overloaded callee candidates.*

$$\begin{aligned} \mathsf{atomic} &:= \mathsf{op_p}\ T \mid T\ \mathsf{op_s} \mid T\ \mathsf{op_i}\ t \mid t\ \mathsf{op_i}\ T \mid T(t^*) \mid T.n \mid T.n(t^*) \mid \mathsf{trait}(T) \mid \mathsf{convertible\_to}(T, t) \\ \mathsf{formula} &:= \mathsf{atomic} \mid (\wedge\ \mathsf{formula}^*) \mid (\vee\ \mathsf{formula}^*) \end{aligned}$$

Example 7. *In real C++ code, the constraint formula* `(T x) { ++x; } && ((T x) { x.f(); } || (T x) { x.g(); })` *consists of three atomic constraints* `(T x) { ++x; }`, `(T x) { x.f(); }`, *and* `(T x) { x.g(); }`. *The overall requirement is that the type* `T` *must support the prefix-increment operator* `++`, *and must support member function calls of either* `f()` *or* `g()`.

We do not reason about implications between different atomic constraints except for equality comparisons. Certain atomic constraints are *normalized* to reduce redundancies: for example, `(T x, T y) { x + y; }` and `(T x, T y) { y + x; }` are treated as the same atomic constraint by rewriting $y+x$ to $x+y$ (i.e. the x is always on the left hand side). Our main reasoning effort is devoted to conjunctions and disjunctions, *e.g.* removing du-

plicate conjuncts. Also, we consider constraint formulas for each template type parameter T, but the formula itself can involve other type parameters, such as `std::same_as<T, U>`, which is an atomic constraint formula requiring T to be the same as another template type parameter U.

6.3.3 Inter-procedural constraint map construction

The first step is to traverse the abstract syntax tree (AST) of the `C++` translation unit and construct the *inter-procedural constraint map*[3] representing both intra-procedural constraints inside individual function templates and the relations between different function templates. The traversal can be either depth-first search or breadth-first search, and we primarily collect all usage sites of variables of types in the template parameter list. Formally, for each `C++` translation unit $\mathcal{U}$, we use $\mathcal{U}_F$ to denote the set of function templates in $\mathcal{U}$. For each function template $f_i \in \mathcal{U}_F$, we use $\mathbf{TTParm}(f_i)$ to denote the set of type template parameters of f_i. Given a translation unit $\mathcal{U}$, we define the *inter-procedural constraint map* $M_{\mathcal{U}}$, which maps each type template parameter in $\bigcup_{f_i \in \mathcal{U}_F} \mathbf{TTParm}(f_i)$ to a set of constraints that it should satisfy.

Definition 18 (Inter-procedural constraint map)**.** *Given a specific type template parameter $T \in \mathbf{TTParm}(f_i)$, we classify the constraints in $M_{\mathcal{U}}(T)$ into two categories.*

- ***Intra-procedural constraints:*** *This category includes constraints on T that are not dependent on other functions or function templates, such as unary/binary operators, member accesses, etc., inside the body of f_i. They are atomic constraints as defined in Definition 17.*

- ***Inter-procedural constraints:*** *An inter-procedural constraint is generated for each named call-site $g(\ldots)$ inside f_i where a variable of type T is used as the k-th argument. Because of overloading, g can refer to many functions or function templates*

[3] The same idea can be extended to class templates, which could be defined as *inter-class constraint map*.

$$\frac{\mathsf{op_p}\ v \quad v:T}{\mathsf{op_p}\ T}\ (1) \quad \frac{v\ \mathsf{op_s}}{T\ \mathsf{op_s}}\ (2) \quad \frac{v\ \mathsf{op_i}\ e \quad v:T \quad e:t}{T\ \mathsf{op_i}\ t}\ (3) \quad \frac{e\ \mathsf{op_i}\ v \quad e:t \quad v:T}{t\ \mathsf{op_i}\ T}\ (4)$$

$$\frac{v(e^*) \quad v:T \quad e^*:t^*}{T(t^*)}\ (5) \quad \frac{v.n \quad v:T}{T.n}\ (6) \quad \frac{v.n(e^*) \quad v:T \quad e^*:t^*}{T.n(t^*)}\ (7) \quad \frac{\mathsf{trait}(T)}{\mathsf{trait}(T)}\ (8)$$

$$\frac{n(\ldots v \ldots) \quad v:T \quad (<\ldots U \ldots> \ \_n(\ldots U \_\ldots)\ \{\ldots\})^* \quad ((<>)?\ \_n(\ldots t \_\ldots)\ \{\ldots\})^*}{[(U,m)^*, t^*]}\ (9)$$

Figure 6.6: Rules for inter-procedural constraint map construction.

sharing the same name: $\{g^0, g^1, \ldots\}$*. The inter-procedural constraint for this call-site is thus a list of elements in the following forms.*

- (U, m) *corresponds to another function template* g^j*, such that the* k*-th parameter of* g^j *is of type* $U \in \mathbf{TTParm}(g^j)$*.* m *is a corresponding* backmap *which will be explained later. Overall, this means* T *should satisfy whatever* U *satisfies.*
- t *corresponds to a function or function template specialization*[4] g^k*, such that the* k*-th parameter of* g^k *is of concrete type* t*; or a function template* g^k *but the* k*-th parameter of* g^k *is of concrete type* t*. Overall, this means* T *should be convertible to* t*.*

The formal constraint collection process is shown in Figure 6.6, where the underscore "_" means "don't care" tokens (i.e., source code tokens that are ignored). Rules (1)-(8) correspond to the intra-procedural uses as defined in Figure 6.5, which generates intra-procedural constraints; rule (9) is the inter-procedural use ($n(\ldots v \ldots)$) as defined in Figure 6.5, which generates inter-procedural constraints. The constraints collected in this stage are not yet in the form of constraint formulas in Definition 17. In the formula map construction stage in Section 6.3.4, (unsimplified) constraint formulas as defined in Definition 17

[4] C++ only allows full specialization for function templates, meaning that every template parameter should be concrete in the specialization.

are generated, where conjunctions are constructed to model multiple requirements in the same function template, and disjunctions are constructed to model function overloading.

To correctly propagate constraints inter-procedurally, we need *backmaps*. We explain the intuition through an example. Consider the following C++ code.

```
template <typename T> void f(T x) { x++; }
template <typename U> void g(U x) { f(x); }
```

We need to propagate the constraint on the template parameter T of f, which is `(T x) { x++; }`, to the template parameter U of g. However, we cannot directly copy that constraint, because at g, there is no type template parameter named T. To resolve this issue, we design backmaps, which store what arguments (in this case, U) are substituted for the callee's type template parameter (in this case, T). In a more general case where we have a chain of function calls (f_1 calls f_2, f_2 calls f_3, etc.) of length l, the correct type can be resolved by iteratively stepping through the l backmaps.

Definition 19 (Backmap)**.** *For each named call-site $g(\ldots)$ inside a function template f, for each function template g^i in the overloading candidate set $\{g^0, g^1, \ldots\}$, the* backmap *m_i is defined as a (possibly non-total) map mapping each type template parameter $U \in \mathbf{TTParm}(g^i)$ to either a concrete type t or a type depending on template parameters in $\mathbf{TTParm}(f)$.*

In our implementation, we construct backmaps in a best-effort fashion. This involves analyzing the named call-site and identifying the arguments that should be used to replace the type template parameters of the callee. If the correct type cannot be resolved, then we simply discard the constraint, which still preserves the under-approximation property.

Example 8 (Inter-procedural constraint map and backmap)**.** *For the following code,*

```
void f(int x) {}
template <typename T> void f(T x) { x++; ++x; x+=1; }
template <typename U> void g(U x) { f(x); x.print(); }
```

the corresponding inter-procedural constraint map is

$$\left\{ \begin{array}{ll} T & \rightarrow \{ \texttt{(T x)\{x++;\}}, \ \texttt{(T x)\{++x;\}}, \ \texttt{(T x,int y)\{x+=y;\}} \} \\ U & \rightarrow \{ [\texttt{int}, (T, m)], \ \texttt{(U x)\{x.print();\}} \} \end{array} \right\}$$

where $[\texttt{int}, (T, m)]$ *is an inter-procedural constraint as described in Definition 18, and the corresponding backmap* m *for the function template* `f` *(not the separate overloading of* `f` *with concrete type* `int`*) is*

$$\{T \rightarrow U\}.$$

Note that in our implementation, the keys of constraint maps are pointers to template type parameters in the AST (`const clang::TemplateTypeParmDecl*`), so there is no ambiguity even if different function templates use the same name (such as T) for their template parameters.

6.3.4 Formula map construction

The inter-procedural constraint map M contains all we need for constraint synthesis, but recursive dependencies could exist. For example, the constraints of a function template's type parameter T can depend on another function template's type parameter U, which can, recursively, depend on T again. Our second step is thus obtaining the *formula map*, which maps type template parameters to constraints formulas, and which does not contain recursive dependencies. Algorithm 4 gives the formula map construction algorithm, which employs a depth-first search on the inter-procedural constraint map.

Algorithm 4 maintains two data structures. The *status* map at Line 3 represents whether the constraint formula of a type template parameter has not been touched by the algorithm (NOTVISITED), is in the process of being constructed (ONSTACK), or has already been constructed (VISITED). The *result* map at Line 5 represents the formula map being computed, wherein the implementation, we actually store the pointers to constraint formulas,

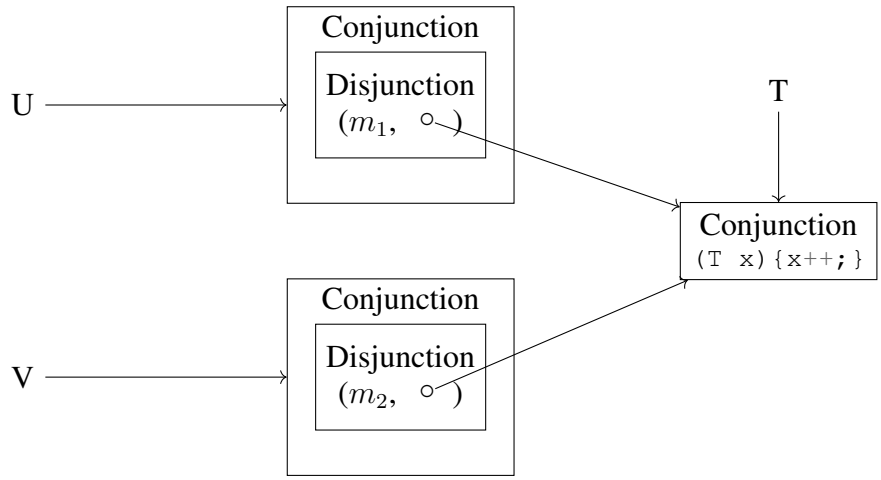
Algorithm 4 The formula map construction algorithm

```
1:  function constructFormulaMap(M /* inter-procedural constraint map */)
2:      // type template parameter → construction status
3:      status ← emptyMap(default = NOTVISITED)
4:      // type template parameter → formula
5:      result ← emptyMap()
6:      function dfs(T /* type template parameter */)
7:          if status[T] = NOTVISITED then
8:              status[T] ← ONSTACK
9:              conjunction ← emptyConjunction()
10:             for constraint ∈ M[T] do
11:                 if constraint.intra() then // intra-procedural constraint
12:                     conjunction.add(constraint)
13:                 else if constraint.inter() then // inter-procedural constraint
14:                     disjunction ← emptyDisjunction()
15:                     for (U, m) ∈ constraint do
16:                         temp ← dfs(U)
17:                         if temp = NULL then
18:                             disjunction.add(true)
19:                         else
20:                             disjunction.add((m, temp))
21:                     for t ∈ constraint do
22:                         disjunction.add(convertible_to(T, t))
23:                     conjunction.add(disjunction)
24:             result[T] ← conjunction
25:             status[T] ← VISITED
26:             return conjunction
27:         else if status[T] = ONSTACK then
28:             return NULL
29:         else if status[T] = VISITED then
30:             return result[T]
31:     for T ∈ M.keys() do
32:         if status(T) = NOTVISITED then
33:             dfs(T)
34:     return result
```

so that the same constraint formula for a type template parameter of a callee can be shared by different callers. The function *DFS* at Line 6 recursively construct the constraint formula for each type template parameter (Line 31), and any recursive dependency in the inter-procedural constraint map is truncated at Line 27 and is eventually treated as `true`

```
template <typename T> void f(T x) { x++; }
template <typename U> void g1(U x) { f(x); // call-site 1 }
template <typename V> void g2(V x) { f(x); // call-site 2 }
```

(a) Three C++ function templates where `g1` and `g2` share the callee `f`.



(b) The corresponding formula map where T, U, V are type template parameters pointing to their constraints.

Figure 6.7: A piece of C++ code and its corresponding formula map.

(Line 18). Inside *DFS*, the only case where we involve backmaps is for inter-procedural constraints of the form (U, m), where we also store the backmap to the constraint formula being constructed at Line 20 so that we can recover type names for inter-procedural constraints.

The order of visiting keys (Line 31) can affect the final results. Consider this chained dependency of type template parameters $T \leftarrow U \leftarrow V \leftarrow T$, meaning that T depends on U, U depends on V, and V depends on T again. If T is visited first, then when we recursively get to V, V's dependency of T will be truncated to `true`; if V is visited first, then V's dependency of T will at least include the intra-procedural constraints of T. This is a loss of precision when handling recursions, and we choose to keep the algorithm lightweight without introducing time-consuming processes such as fixed point computations.

Example 9 (Formula map)**.** *For the C++ code shown in Figure 6.7a, the corresponding formula map is shown in Figure 6.7b, where* m_1 *and* m_2 *are backmaps for call-site* 1 *and*

call-site 2*, respectively. The constraint of* T *is also shared inside* U *and* V*'s constraints, because* `g1` *and* `g2` *both call* `f`*. Note that there are redundant nested conjunctions and disjunctions in this case, which can be handled by our formula simplification algorithm in Section 6.3.5.*

6.3.5 Formula simplification

From the formula map, we can recursively read and print the constraint formula (as defined in Definition 17) for each type template parameter, with the help of backmaps. However, because of chained function calls in function templates, the constraint formulas could contain redundancies such as $(\wedge(\vee(\wedge(\vee(\mathsf{atomicConstraint})))))$. A typical case is that when there is only one callee for a call-site inside the template body, Algorithm 4 still insert a disjunction layer in the formula. To make the final constraint formula smaller, we apply a simple polynomial-time simplification process (Algorithm 5) before actually inserting the constraint formula into `C++` code.

The basic idea of Algorithm 5 is to simplify conjunctions (Line 1) and disjunctions (Line 27) in a bottom-up fashion using recursion. Redundant layers of the formula are recursively eliminated (Line 7). The main simplification rules include de-duplicating terms and discarding trivial terms (Line 12). The main procedure (Line 29) starts the simplification based on the input formula's format (atomic, conjunction, disjunction).

There exist sophisticated Boolean formula minimization algorithms, such as the Quine-McCluskey algorithm [117, 118, 119], but the problem itself is NP-hard [120]. Since our goal is to keep our approach lightweight, we choose to use our Algorithm 5. Suppose the input formula f's length is l and nesting depth is d, and suppose the hash function takes linear time with respect to the input length. Then Algorithm 5's time complexity is $O(ld)$, and is thus polynomial time with respect to the formula size.

6.3.6 Soundness versus soundiness

Algorithm 5 The formula simplification algorithm

```
1: function SIMPLIFYCONJUNCTION(f /* constraint formula */)
2:     newConjunction ← emptyConjunction()
3:     deduplicate ← emptyHashSet()
4:     for conjunct ∈ f do
5:         candidates ← emptyList()
6:         conjunct ← SIMPLIFYFORMULA(conjunct) // recursive simplification
7:         if conjunct.isConjunction() then // inspect one layer of conjunction
8:             for child ∈ conjunct do
9:                 candidates.add(child)
10:        else
11:            candidates.add(conjunct)
12:        for candidate ∈ candidates do
13:            if candidate.isTrue() then // omit trivial conjuncts
14:                continue
15:            else if candidate.isFalse() then // f is trivially false
16:                return false
17:            else
18:                if not deduplicate.contains(candidate.hash()) then
19:                    newConjunction.add(candidate)
20:                    deduplicate.add(candidate.hash())
21:    if newConjunction.length() = 0 then // f is trivially true
22:        return true
23:    else if newConjunction.length() = 1 then // omit one trivial conjunction layer
24:        return newCojunction.first()
25:    else
26:        return newCojunction
27: function SIMPLIFYDISJUNCTION(f /* constraint formula */)
28:    // similar to SIMPLIFYCONJUNCTION, omitted
29: function SIMPLIFYFORMULA(f /* constraint formula */)
30:    if f.isAtomic() then
31:        return f
32:    else if f.isConjunction() then
33:        return SIMPLIFYCONJUNCTION(f)
34:    else// disjunction
35:        return SIMPLIFYDISJUNCTION(f)
```

Our goal is to let the computed constraint formula $\mathcal{C}$ under-approximate (allow more types than) the real requirement $\mathcal{R}$. Suppose the set of types (including new types that can be added to the code in the future) allowed by $\mathcal{C}$ and $\mathcal{R}$ are denoted as $S(\mathcal{C})$ and $S(\mathcal{R})$, respectively; as we discussed in Section 6.2, we need $S(\mathcal{C}) \supseteq S(\mathcal{R})$. We call this property

```cpp
#include <utility>

struct S { void f() && {} };

template <typename T>
void g(T x) { std::move(x).f(); }

int main() { g(S{}); }
```

(a) Original code: successful compilation on Apple clang 15.0.0.

```cpp
#include <utility>

struct S { void f() && {} };

template <typename T>
requires requires (T o) { o.f(); }
void g(T x) { std::move(x).f(); }

int main() { g(S{}); }
```

(b) Modified code: failed compilation on Apple clang 15.0.0.

Figure 6.8: An unsound corner case where our tool inserted over-constrained constraints. The member function `f` of `S` should be invoked on r-values, while our tool ignores references and uses an l-value to invoke `f` in the constraint. Note that **`requires requires`** is not a typo: the first **`requires`** specifies the constraint for the template while the second **`requires`** starts a constraint expression.

soundness. In our simplified calculus (Figure 6.5), soundness is ensured by our algorithms.

Theorem 17. *For programs written in the simplified calculus shown in Figure 6.5, for every type template parameter T, if a type argument t is passed in for T and does not result in compile-time errors, then the constraint formula generated for T according to the three steps (Section 6.3.3, Section 6.3.4, Section 6.3.5) evaluates to true on t.*

Proof. Since the type argument t does not result in compile-time errors, all uses of type t in the code are valid. Every intra-procedural constraint as defined in Definition 18 is satisfied, so the conjunction of these constraints is also satisfied. Every inter-procedural constraint, as defined in Definition 18, according to overloading resolution, should result in at least one valid candidate, so the disjunction of the constraint formulas from the overloading candidates is satisfied. The choice of truncating recursive dependencies to `true` in Section 6.3.4 only relaxes the constraint, so satisfaction is preserved. The formula simplification algorithm as described in Section 6.3.5 preserves the truth value of the constraint formulas, so the simplified formula is also satisfied by t. □

However, almost all realistic static analysis tools are unsound in certain aspects [121]. The reasons include scalability, precision, and engineering details of realistic program-

ming languages. A static analysis tool is *soundy* when most common language features are soundly-approximated while some special language features, well-known to experts in the area, are unsoundly-approximated [121]. This is known as the *soundiness*. We claim our implementation is soundy for `C++`. First, we apply the following general strategy in our implementation: whenever we encounter unsupported `C++` language features, we treat the constraint generated by the unsupported part as `true`, so our tool gracefully bypasses them and approximates toward $S(\mathcal{C}) \supseteq S(\mathcal{R})$. Second, for cv(const/volatile)-qualifiers and lvalue/rvalue references, with deliberately designed corner cases such as Figure 6.8, our tool can generate slightly over-constrained constraints and therefore results in $S(\mathcal{C}) \subset S(\mathcal{R})$. In Figure 6.8, the reason is that we ignore references when handling constraints. To sum up, our tool is sound except for the two features (1) cv-qualifiers and (2) lvalue/rvalue references. Those two features have complicated interactions with template argument deductions [122, 123], and we leave the completely sound handling of them as future work.

6.4 Experiments

We implemented our approach based on the `RecursiveASTVisitor` facility from Clang frontend (forked from the main branch of Clang in October 2023). Our implementation language is `C++`, consisting of roughly 2.8 kLOC. We support unary operators, binary operators, higher-order functions, class member accesses, and simple type traits as atomic constraints. We will open-source our implementation and provide the link in the final version of this chapter.

We conducted experiments on two libraries: `<algorithm>` from the Standard Template Library (STL) and `<boost/math/special_functions.hpp>` from the Boost library. We chose these two libraries because they mainly consist of function templates instead of class templates, and our tool currently only targets function templates, although the idea can also be extended to class templates. Our evaluation focuses on three dimensions.

- *Performance (Section 6.4.1).* For both libraries, we report the numbers of function

Table 6.1: Overall performance. The execution time includes not only the three steps described in Section 6.3, but also the actions of generating the rewritten code, reporting statistical results, etc.

	`algorithm`	`special_functions`
Total LOC after preprocessing	32,206	111,862
Execution time (seconds)	0.250	1.302
Number of templates	821	2,531
Number of templates with nontrivial results	315	908

templates for which our tool can infer nontrivial constraints (constraints that are not simply the literal `true`). We also measure execution time.

- *Precision (Section 6.4.2).* For the `algorithm` library, we compare the inferred constraints with the documented constraints.

- *Error reduction (Section 6.4.3).* For both of the `algorithm` library and the `special_functions` library, we measure the Clang compilation error message reductions after adding our constraints.

It is important to mention that our tool is lightweight and can be used on ordinary hardware. To demonstrate this, all of our experiments were conducted on a MacBook Air (2020) with Apple M1 chip and 8GB memory, and everything was executed in a single thread. The pre-processing of standard library headers and the compilation error measurements were all based on the main compiler on the MacBook, which is Apple Clang version 15.0.0.

6.4.1 Overall performance

Table 6.1 presents the execution speed and the number of function templates with nontrivial synthesized constraints. Our analysis is highly efficient, taking only 0.250 seconds to process more than 30k lines of code from `algorithm` and only 1.302 seconds to process over 110k lines of code from `special_functions`. Note that these times include not only the three steps described in Section 6.3, but also the reporting of statistical results, and

```
template <typename _Integral>
__attribute__((__visibility__("hidden")))
__attribute__((__exclude_from_explicit_instantiation__))
__attribute__((__abi_tag__("v160006"))) constexpr
typename enable_if
<
    is_integral<_Integral>::value,
    _Integral
>::type
__half_positive(_Integral __value)
{
    return static_cast<_Integral>(
        static_cast<__make_unsigned_t<_Integral> >(__value) / 2
    );
}
```

Figure 6.9: An example function template that our tool didn't synthesize constraints.

the code rewriting for adding synthesized constraints. This demonstrates the exceptional performance of our tool. Furthermore, our tool can synthesize non-trivial constraints for approximately 30%-40% of function templates. The remaining function templates either lack requirements in our supported categories of atomic constraints, or incorporate C++ features that are not yet supported by our tool, such as complicated type sugars or aliases. An example from the `algorithm` library that our tool didn't synthesize constraints is shown in Figure 6.9, where the usage of the variable `__value` is wrapped in type conversions before being divided by the operator `/`, and our tool currently only supports limited forms of such wrappers. It is worth mentioning that our conservative strategy (Section 6.3.6) ensures under-approximation by treating an inter-procedural constraint as `true` when there exists a callee candidate containing unsupported features. This can result in some function templates having trivial constraints.

Summary. Our analysis is highly efficient and can synthesize non-trivial constraints for 30%-40% of function templates. In particular, the high efficiency makes it possible to use our tool in interactive settings where the developer is editing the code.

Table 6.2: Precision on STL `algorithm` library. The library requirement is obtained from the standard document, while the synthesized requirement is generated by our tool.

Template	Documented requirement	Synthesized requirement
`for_each`	(InputIterator, UnaryFunction)	(Iterator, UnaryFunction)
`find`	(InputIterator, Any)	(Iterator, Any)
`find_if`	(InputIterator, Predicate)	(Iterator, Predicate)
`count`	(InputIterator, Any)	(Iterator, Any)
`count_if`	(InputIterator, Predicate)	(Iterator, Predicate)
`replace`	(ForwardIterator, Any)	(Iterator, Any)
`replace_if`	(ForwardIterator, Predicate, Any)	(Iterator, Predicate, Any)
`copy`	(InputIterator, OutputIterator)	(Any, Any)
`copy_if`	(InputIterator, OutputIterator, Predicate)	(Iterator, Iterator, Predicate)
`move`	(InputIterator, OutputIterator)	(Any, Any)
`unique_copy`	(InputIterator, OutputIterator)	(Iterator, Iterator)
`sort`	(ValueSwappable $\wedge$ RandomAccessIterator)	(RandomAccessIterator)
`equal_range`	(ForwardIterator, Any)	(Iterator, Any)
`merge`	(InputIterator, InputIterator, OutputIterator)	(Iterator, Iterator, Iterator)

6.4.2 Precision on `algorithm`

To further understand the quality of the synthesized non-trivial constraints for `algorithm`, we conduct a semi-automated comparison of our generated requirements with the ones documented in the `C++` standard. We choose the 14 function templates from the introductory `C++` textbook [113] as our targets.

The measurement process is as follows. First, we collect a set of representative *named requirements* [124] that are used in the `C++` standard to specify the expectations of template parameters in the standard library. We create a constraint formula evaluator, which incorporates certain named requirements as hard-coded components. The evaluator then runs on the synthesized constraint formula and reports the most general named requirement N that evaluates to true on the formula. This implies that N is at least as constrained as the formula itself.[5] Next, we manually compare the generated named requirements with the ones described in the documentation.

[5]This method can, in general, be used for matching constraints with pre-defined concepts, as discussed in detail in Section 6.6.1.

Table 6.2 shows the library requirements obtained from the document and the synthesized requirements obtained from the above process. We can observe that the synthesized requirements always under-approximate the library requirements, and they are often similar or identical to the documented requirements. For the first type template parameter of `for_each`, the library requirement is `InputIterator`, while the generated constraint formula is

```
(requires (T x0) { *x0; } && requires (T x0) { ++x0; })
```

which is inferred to be just `Iterator` by our formula evaluator. The distinction between these types of iterators can be complex [125], involving details that our tool does not handle. However, the conclusion of `Iterator` already provides more information than the unconstrained case, and in clearer cases, our tool can also synthesize more precise iterators such as the `RandomAccessIterator` for `sort`, where the code of `sort` provides enough information for our tool to differentiate it from normal iterators. For `copy` and `move` (not to be confused with the `move` for move semantics), our tool synthesized trivial constraints. This is because in the version of STL targeted by our experiment, they were implemented using a helper `struct _ClassicAlgPolicy`, which our tool currently does not handle `struct` or `class`.

Summary. Our tool can synthesize meaningful constraints under-approximating the standard library document for `algorithm`, and in many cases the generated constraints are the same as the library requirements. This demonstrates that our tool can synthesize constraints improving the template interface's clarity.

6.4.3 Error message reduction on `algorithm` and `special_functions`

We also examine the reduction in error messages when incorrect arguments are used for both `algorithm` and `special_functions`. Specifically, for each function template `f`, we use a Python script to introduce a new empty `struct S {};` along with a variable `s` of

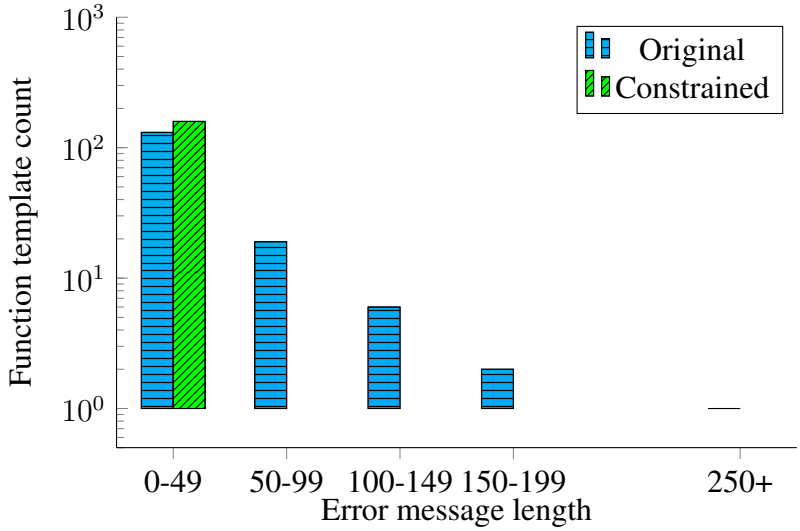


Figure 6.10: Error message length (number of lines) distribution on `algorithm`. The y-axis is of logarithm scale, so most lengths reside in $[0, 50)$. The average lengths are 30.022 for the original code and 13.019 for the constrained code.

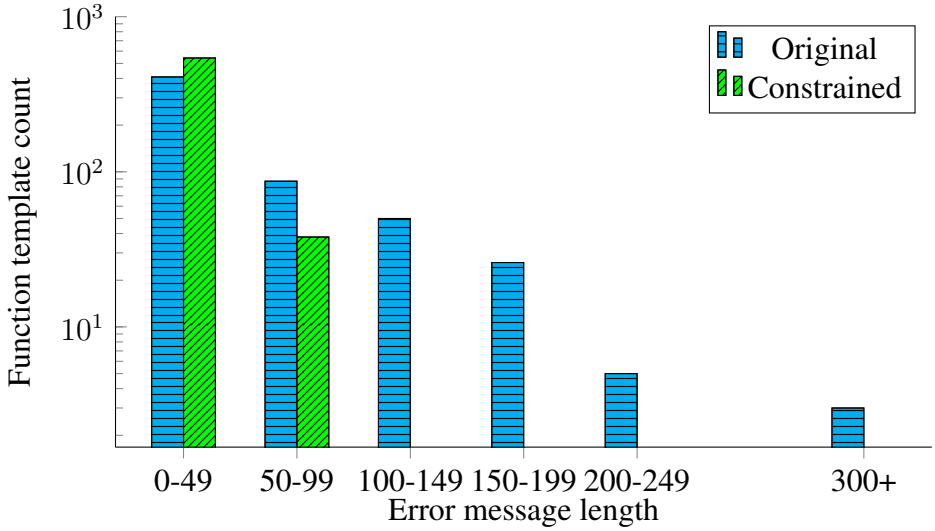


Figure 6.11: Error message length (number of lines) distribution on `special_functions`. The y-axis is of logarithm scale, so most lengths reside in $[0, 50)$. The average lengths are 43.769 for the original code and 15.826 for the constrained code.

type `struct S` into the code. We then make a function call `f(s, s, ...)`. Given that the new type `S` is unlikely to satisfy the requirements of `f`, template-related error messages are expected. We compare the lengths (numbers of lines) of error messages before and after

the addition of the constraints. Figure 6.10 and Figure 6.11 give the distribution of these lengths. Note that the Boost library `special_functions` includes some STL headers as well, and for measurements of the error message reductions in this case, we excluded STL function templates and only considered Boost function templates.

From these two figures, it is clear that the synthesized concepts can successfully intercept the errors at the early stages and greatly reduce the length of error messages. Constrained templates consistently produce error messages that are less than 100 lines for both libraries. This provides strong evidence that the new errors are more comprehensible and manageable.

Summary. The constraints synthesized by our tool for `algorithm` and `special_functions` can effectively reduce the lengths of error messages. Additionally, even for shorter error messages, the constrained version could be more comprehensible, as discussed in Section 6.5.2.

6.5 Case studies

To better understand the error message improvements, we conduct three case studies. In Section 6.4, our tool synthesized constraints for `<algorithm>` from the Standard Template Library (STL) and `<boost/math/special_functions.hpp>` from the Boost library. We select two incorrect `C++` programs using STL and one incorrect `C++` program using Boost, and study the error message improvements after adding the synthesized constraints to STL/Boost. The incorrect programs using STL are two real-world examples obtained from the StackOverflow Q&A website, and the incorrect program using Boost is one obtained from our synthetic incorrect programs in Section 6.4.3. All errors were produced by Apple Clang version 15.0.0.

6.5.1 Case 1: Real-world error example of `std::binary_search` from StackOverflow

```
In file included from test1.cc:1:
In file included from /Library/Developer/CommandLineTools/SDKs/...
In file included from /Library/Developer/CommandLineTools/SDKs/...
...failed due to requirement '__is_callable<(lambda at test1.cc...
  static_assert(__is_callable<_Compare, decltype(*__first), const...
  ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~...
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
    __first = std::lower_bound<_ForwardIterator, _Tp, __comp_ref...
                  ^
test1.cc:16:34: note: in instantiation of function template spec...
    bool isElementPresent = std::binary_search(
                                 ^
In file included from test1.cc:1:
In file included from /Library/Developer/CommandLineTools/SDKs/...
In file included from /Library/Developer/CommandLineTools/SDKs/...
...lower_bound.h:40:9: error: attempt to use a deleted function
    if (std::__invoke(__comp, std::__invoke(__proj, *__m), __...
        ^
...specialization 'std::__lower_bound_impl<std::_ClassicAlgPolicy...
  return std::__lower_bound_impl<_ClassicAlgPolicy>(__first, __...
             ^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
(20 lines omitted)
3 errors generated.
```

(a) Error messages of the template `std::binary_search` before adding constraints.

```
...:16:29: error: no matching function for call to 'binary_search'
    bool isElementPresent = std::binary_search(
                            ^~~~~~~~~~~~~~~~~~
...candidate template ignored: constraints not satisfied [with...
binary_search(_ForwardIterator __first, _ForwardIterator __last,...
^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
...requires (_Compare f, _Tp x0, _ForwardIterator x1) { f(x0, *x1); }
                                                                ^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
binary_search(_ForwardIterator __first, _ForwardIterator __last,...
^
1 error generated.
```

(b) Error messages of the template `std::binary_search` after adding constraints.

Figure 6.12: Error messages comparison on `std::binary_search`.

This case study is based on a real-world question from StackOverflow [126]. The question is about a compilation error on `C++` code with an incorrect use of `std::binary_search`

on a vector of a custom class: the provided lambda expression only accepts objects of the custom class while the target of the binary search is an `int`. Our tool synthesized constraints for `std::binary_search` so the error message for that code can be improved. The effective part of the added constraint is as follows.

```
requires (_Compare f, _Tp x0, _ForwardIterator x1) { f(x0, *x1); }
```

The error messages before/after adding the constraint is shown in Figure 6.12. Note that in the error messages before adding the constraint, there are 20 omitted lines, which matches Section 6.4.3's conclusion that adding constraints can effectively reduce the error message length. Also, in the error messages before adding the constraint, there are unnecessary implementation details such as `std::__lower_bound_impl<_ClassicAlgPolicy>`, while in the error messages after adding the constraint a clear explanation of the compilation error "`constraints not satisfied`."

6.5.2 Case 2: Real-world error example of `std::sort` from StackOverflow

This case study is based on a real-world question from StackOverflow [127]. The question is about a compilation error on `C++` code with the incorrect use of `std::sort` on an array of a custom class. `std::sort` requires a pair of pointers, while the incorrect code uses a pair of custom objects. Our tool synthesized constraints for `std::sort` so the error message for that code can be improved. The effective part of the added constraint is as follows.

```
requires
(
 requires (_RandomAccessIterator x0) { *x0; } &&
```

```
(135 lines omitted)
...: note: in instantiation of...
    std::sort(graph->edge[0], graph->edge[(graph->E)-1], myComp<int>);
         ^
...: error: no type named...
  typedef typename iterator_traits<_RandomAccessIterator>::...
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~...
...: note: in instantiation of...
    std::__sort<_WrappedComp>(std::__unwrap_iter(__first), ...
         ^
...: error: invalid operands...
  difference_type __depth_limit = 2 * __log2i(__last - __first);
                                      ~~~~~~ ^ ~~~~~~~
...: note: candidate template ignored:...
operator-(const reverse_iterator<_Iter1>& __x, const...
^
12 warnings and 8 errors generated.
```

(a) Error messages of the template `std::sort` before adding constraints.

```
(33 lines omitted)
...: warning: user-defined literal suffixes not starting with '_'...
    __attribute__((__visibility__("hidden")))...
...^
...: error: no matching function for call to 'sort'
    std::sort(graph->edge[0], graph->edge[(graph->E)-1], myComp<int>);
    ^~~~~~~~~
...: note: candidate template ignored: constraints not satisfied...
void sort(_RandomAccessIterator __first, _RandomAccessIterator...
     ^
...: note: because '*x0' would be invalid...
 requires (_RandomAccessIterator x0) { *x0; } &&
                                       ^
...: note: candidate function template not viable:...
void sort(_RandomAccessIterator __first, _RandomAccessIterator...
     ^
12 warnings and 1 error generated.
```

(b) Error messages of the template `std::sort` after adding constraints.

Figure 6.13: Error messages comparison on `std::sort`.

```
 requires (_RandomAccessIterator x0) { ++x0; } &&

 requires (_RandomAccessIterator x0) { --x0; }

)
```

The error messages before/after adding the constraint are shown in Figure 6.13. Before adding the constraints, there are 151 lines of warnings and errors generated (where the warnings are related to C++ preprocessing in our experimental steps and are irrelevant to the main errors), where there are 8 errors in total. After adding the constraints, the number of lines of warnings and errors is reduced to 49, and the number of errors is reduced to 1. Again, the new error message provides a clear explanation of the reason: "`constraints not satisfied`."

6.5.3 Case 3: Synthetic error example of `boost::math::sign` from our experiments

This case study is based on one of the synthetic erroneous code described in Section 6.4.3. The code makes use of `boost::math::sign`, but the argument types are incorrect, resulting in compilation errors. Our tool synthesized constraints for `boost::math::sign`. The effective part of the added constraint is as follows.

```
requires (T x0, int x1) { x0 == x1; }
```

The error messages before/after adding the constraint are shown in Figure 6.14. Although, in this case, the lengths of error messages before/after adding the constraint are similar, it is arguably true that the messages after adding the constraint are clearer and easier to understand. Specifically, before adding the constraint, the implementation details of the function template are exposed (`return (z == 0) ? 0 : (boost::math::signbit)(z) ? -1 : 1;`). After adding the constraint, it becomes evident that the template is ignored due to "`constraints not satisfied`" and "`because 'x0 == x1' would be invalid.`"

```
<stdin>:76087:14: error: invalid operands to binary expression
('const S' and 'int')
   return (z == 0) ? 0 : (boost::math::signbit)(z) ? -1 : 1;
           ~ ^  ~
<stdin>:111864:14: note: in instantiation of function template
specialization 'boost::math::sign<S>' requested here
boost::math::sign(s);
             ^
1 error generated.
```

(a) Error messages of the template `boost::math::sign` before adding constraints.

```
<stdin>:126854:1: error: no matching function for call to 'sign'
boost::math::sign(s);
^~~~~~~~~~~~~~~~
<stdin>:80713:12: note: candidate template ignored:
constraints not satisfied [with T = S]
inline int sign (const T& z)
           ^
<stdin>:80712:30: note: because 'x0 == x1' would be invalid:
invalid operands to binary expression ('S' and 'int')
requires (T x0, int x1) { x0 == x1; }
                             ^
1 error generated.
```

(b) Error messages of the template `boost::math::sign` after adding constraints.

Figure 6.14: Error messages comparison on `boost::math::sign`.

6.6 Discussions

6.6.1 Matching with pre-defined concepts

Our approach synthesizes a constraint formula for each type template parameter. However, there are also pre-defined concepts such as `std::integral` or domain-specific ones created by programmers. Our approach can be easily extended to match the synthesized constraint formula with these pre-defined concepts as explained in Section 6.4.2, using evaluators on the constraint formulas. Specifically, we view the pre-defined concept as a predicate P, and suppose the synthesized constraint formula is of the form $((A \vee B) \wedge C)$. We first check whether P implies atomic components A, B, C, and then combine them using $\vee$ or

$\wedge$. This idea for comparing predicates is also general and can potentially have other usage scenarios.

6.6.2 Generalization

Our approach targets C++ templates, but the general idea is applicable to various programming languages in different usage scenarios. For example, disjunctions can model the requirements of various kinds of polymorphisms, such as overloading, dynamic dispatch, etc. The backmap Definition 19 idea can, in general, retain different information as needed at call-sites. Our goal of "synthesizing constraints for C++ templates" can also be abstracted to "synthesizing requirements for functions," so it shares similarities with API/specification inference techniques [128]. The paper focuses on C++ as a concrete and tangible illustration of the overarching idea.

6.6.3 Higher-level semantics of programming languages

Our work also advocates attention to high-level semantics of programming languages. In particular, C++ templates themselves are not translated into middle-level IR or assembly code by the compiler: only template instantiations are preserved. Indeed, the main problem we target (compilation error) is no longer accessible in middle-level IR, such as LLVM-IR [129]. To deal with such high-level semantics, it is necessary to confront complicated high-level program representations. In our case, we directly analyze the C++ AST.

6.6.4 Usage scenarios

We expect our tool to be used mainly in two scenarios. First, our tool can be applied to existing templated C++ code, which can improve the interface and serve as a precaution for future error messages. Second, because of the high analysis speed (Section 6.4.1), our tool can be integrated into IDEs (similar to refactoring tools) and thus provide interactive feedback even when the developers are changing the code. In both usage scenarios, the de-

veloper can choose to either accept or reject the synthesized constraints to ensure absolute soundness.

6.7 Related work

This work attacks a relatively unexplored problem: C++ template constraint synthesis. Our work improves the readability and maintainability of C++ code by leveraging the features of C++20. Our novel technical insight includes using lightweight static analysis to handle complicated programming languages like C++ in real-world scenarios while still ensuring high rigor and soundiness (Section 6.3.6). This section focuses on surveying related topics.

Our work shares similarities with type inference for dynamically-typed languages like JavaScript [110] and Python [111, 112]. However, as highlighted in the introduction, our work does not infer types but infers constraints corresponding to sets of types, synthesizes complicated constraints applicable to newly defined types, and can be regarded as meta-analysis of the compilation process for the statically-typed language C++. There are also previous efforts targeting Java wildcard inference [130, 131]. C++ constraints and concepts can be regarded as a flexible way to specify use-site constraints, which includes use-site covariance/contravariance as special cases.

Existing static analysis work for C++ typically focuses on traditional and general topics such as symbolic execution [132, 133] and model checking [134, 135, 136, 137], which rarely targets high-level C++ semantic problems. In contrast, our work handles recent language features (constraints and concepts) introduced in C++20. While there are existing studies on the formal semantic analysis of C++ templates [138] and comparisons with other languages like Haskell [139], our work presents a practical application for C++ templates. There also exists work handling high-level semantics such as Java reflection [140] and containers [141], while our work focuses on C++ generic programming. There also exists work improving type error messages by using the type checker as an oracle to search for similar programs that do type-check [142]. Our work adds "specifications" to the original

programs, which itself is beneficial for a clearer interface even if no error occurs.

Our work also shares similarities with API/specification [128] inference techniques, but we focus on precise type requirements, while existing API/specification inference techniques [143, 144, 145, 146, 147] utilize data-mining, probabilistic methods, or various heuristics and are thus not inferring precise specifications.

The idea of backmap (Section 6.3) can be regarded as a form of compile-time context-sensitivity, similar to the usual run-time context-sensitivity [148, 149, 150, 151, 152]. In our case, the same function template can be called inside different function templates, each of which passes potentially different "type-contexts" into the callees, which are all resolved in compile time.

The field of program synthesis [153, 154, 155, 156] has achieved great progress in recent years. Our work does not synthesize programs but constraints for template parameters in existing code. Thus, our work targets a different problem and is more scalable (Section 6.4.1) than typical program synthesis techniques.

6.8 Chapter Conclusion

We proposed a framework for automatically synthesizing constraints for `C++` function templates. The synthesized requirements are expressed using `C++20` constraints and concepts, which can significantly improve interface clarity and template-related compilation error messages. Our tool runs fast and thus has the potential to be used in IDE for interactive error reporting. Experimental results showed that our tool can synthesize valid constraints for real-world `C++` code from the Standard Template Library and the Boost library.

CHAPTER 7
CONCLUSION

Witness functions are implicit functions inside computability/complexity theoretic impossibility proofs. The properties of witness functions have implications on (1) decidable approximations of undecidable problems (which is the essential idea behind automatic program analysis and verification techniques) and (2) complexity class separation proofs (including open problems in complexity theory). We proved theoretical results regarding the properties of witness functions in the two cases, which provided new angles to understand program analysis theory and complexity theory. We also discussed two practical program analysis techniques, and showed the way to interpret the witnessability results under these realistic settings.

7.1 Future Directions

We list some future directions for the four pieces of work discussed in previous chapters.

7.1.1 Witness Functions for Undecidable Problems

From the angle of program analysis and verification, witnessable problems are the class of problems admitting computable precision improvements on decidable approximations. Future studies can try to determine if-and-only-if conditions for an undecidable problem to be witnessable, which could provide a more complete characterization of witnessability. Similar if-and-only-if conditions exist for productive sets in computability theory [15], which admits partial computable productive functions (similar to witness functions) for computably enumerable subsets.

On the practical side, as mentioned in Section 3.3, it is computable to decide whether the computed witness is a false positive or false negative. Thus the witnessability results

can potentially help to tackle the test oracle problem [157, 158] in software testing: if the witness can be regarded as a test case for program analyzers/verifiers, then the ground truth of whether it is in the undecidable set of program semantics is computable.

7.1.2 Witness Functions for Complexity Classes

The concepts "artificial problems" (problems appearing in the constructions of time / space hierarchy theorems) and "natural problems" (problems naturally occurring in realistic settings) are not formally defined. Future work can consider clearly defined characteristics of these two classes of problems. Combining with our results, this may shed light on the way to achieve separations for natural problems or the way to prove independence results.

Also, analyzing existing unconditional lower bounds as case studies could enable a deeper understanding of our results. Perhaps the most well-known unconditional lower bounds are from time / space hierarchy theorems [17, 18]. Other examples include the quadratic lower bound of palindrome recognition on single-tape Turing machines, which can be obtained by the crossing sequence argument.

7.1.3 Mutual Refinement

Future studies could try to incorporate non-CFL reachability into the mutual refinement framework, which can potentially bring more precision improvements. In general, the idea of mutual refinement works for algorithms traversing all contributing edges, where the traversed edges can be recorded as the starting point for the next iteration of refinement, under possibly different algorithms.

Also, according to witnessability, we can write programs implementing the computable witness function to compute counterexamples for specific mutual refinement algorithms. Although the counterexamples are typically large and involved, by using techniques such as test case reductions [159] we can possibly get smaller counterexamples and derive hints on further improvements of mutual refinement.

7.1.4 C++ Template Constraint Analysis

Templates are used to achieve generic programming in C++, and template constraints pose additional restrictions on the type parameters of generics. The idea behind our technique is a general intuition for generic programming, which is not limited to C++. Thus a natural future direction is to consider constrained generics in other languages, such as wildcards in Java.

Similar to the mutual refinement case, we can implement computable witness functions to compute counterexamples, and use test case reductions or similar techniques to get useful hints on further precision improvements.

REFERENCES

[1] N. Cutland, *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980, ISBN: 9780521294652.

[2] M. Sipser, "Introduction to the theory of computation," 2012.

[3] W. Landi, "Undecidability of static analysis," *LOPLAS*, vol. 1, no. 4, pp. 323–337, 1992.

[4] T. W. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, 2000.

[5] P. A. Abdulla and B. Jonsson, "Undecidable verification problems for programs with unreliable channels," *Inf. Comput.*, vol. 130, no. 1, pp. 71–90, 1996.

[6] C. Dima and F. L. Tiplea, "Model-checking ATL under imperfect information and perfect recall semantics is undecidable," *CoRR*, vol. abs/1102.4225, 2011.

[7] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[8] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[9] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, 1977, pp. 238–252.

[10] G. A. Kildall, "A unified approach to global program optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, P. C. Fischer and J. D. Ullman, Eds., ACM Press, 1973, pp. 194–206.

[11] J. Z. S. Hu and O. Lhoták, "Undecidability of $d_{j:}$ and its decidable fragments," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 9:1–9:30, 2020.

[12] T. P. Baker, J. Gill, and R. Solovay, "Relativizations of the P =? NP question," *SIAM J. Comput.*, vol. 4, no. 4, pp. 431–442, 1975.

[13] D. Joseph and P. Young, "Some remarks on witness functions for nonpolynomial and noncomplete sets in np," *Theoretical Computer Science*, vol. 39, pp. 225–237, 1985.

[14] D. Kozen, "Indexings of subrecursive classes," *Theor. Comput. Sci.*, vol. 11, pp. 277–301, 1980.

[15] R. I. Soare, *Turing computability: Theory and applications*. Springer, 2016, vol. 300.

[16] R. Soare, *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets* (Perspectives in Mathematical Logic). Springer Berlin Heidelberg, 1999, ISBN: 9783540152996.

[17] J. Hartmanis and R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the American Mathematical Society*, vol. 117, pp. 285–306, 1965.

[18] M. Sipser, *Introduction to the Theory of Computation* (Introduction to the Theory of Computation). Cengage Learning, 2012, ISBN: 9781133187813.

[19] T. L. Veldhuizen, "C++ templates are turing complete," Indiana University, Tech. Rep., 2003.

[20] T. Jech, *Set Theory: The Third Millennium Edition, revised and expanded* (Springer Monographs in Mathematics). Springer Berlin Heidelberg, 2013, ISBN: 9783642078996.

[21] A. Asperti, "The intensional content of rice's theorem," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, 2008, pp. 113–119.

[22] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i," in 1, vol. 38, Springer, 1931, pp. 173–198.

[23] S. Liang, W. Sun, and M. Might, "Fast flow analysis with godel hashes," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, IEEE Computer Society, 2014, pp. 225–234.

[24] G. Hardy, *An introduction to the theory of numbers*. Oxford Science Publication, 1979.

[25] H. Rogers, "Gödel numberings of partial recursive functions," *The journal of symbolic logic*, vol. 23, no. 3, pp. 331–341, 1958.

[26] J. B. Wells, "Typability and type checking in system F are equivalent and undecidable," *Ann. Pure Appl. Log.*, vol. 98, no. 1-3, pp. 111–156, 1999.

[27] B. C. Pierce, "Bounded quantification is undecidable," in *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992, ACM Press, 1992, pp. 305–315.

[28] M. Brown and J. Palsberg, "Breaking through the normalization barrier: A self-interpreter for f-omega," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, ACM, 2016, pp. 5–17.

[29] HaskellWiki, *Ghc/ghci*, https://wiki.haskell.org/GHC/GHCi, Accessed in May 2024.

[30] S. Ding and Q. Zhang, "The normalization barrier revisited," in *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday*, 2024, pp. 1–4.

[31] J. D. Day, V. Ganesh, P. He, F. Manea, and D. Nowotka, "The satisfiability of word equations: Decidable and undecidable theories," in *Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11123, Springer, 2018, pp. 15–29.

[32] M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Decidability and undecidability results for nelson-oppen and rewrite-based decision procedures," in *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4130, Springer, 2006, pp. 513–527.

[33] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1855, Springer, 2000, pp. 154–169.

[34] E. L. Post, "A variant of a recursively unsolvable problem," *Bulletin of the American Mathematical Society*, vol. 52, no. 4, pp. 264–268, 1946.

[35] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation* (Prentice-Hall international series in computer science). Prentice Hall, 1993, ISBN: 9780130202499.

[36] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard, "Word equations with length constraints: What's decidable?" In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7857, Springer, 2012, pp. 209–226.

[37] J. Myhill, "Creative sets," *Journal of Symbolic Logic*, vol. 22, no. 1, 1957.

[38] E. L. Post, "Recursively enumerable sets of positive integers and their decision problems," *Bulletin of the American Mathematical Society*, vol. 50, no. 5, pp. 284–316, 1944.

[39] M. Flatt and PLT, "Reference: Racket," PLT Design Inc., Tech. Rep. PLT-TR-2010-1, 2010, https://racket-lang.org/tr1/.

[40] R. Bruni, R. Giacobazzi, R. Gori, I. Garcia-Contreras, and D. Pavlovic, "Abstract extensionality: On the properties of incomplete abstract interpretations," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 28:1–28:28, 2020.

[41] J. R. Shoenfield, "On degrees of unsolvability," *Annals of mathematics*, pp. 644–653, 1959.

[42] J. Moyen and J. G. Simonsen, "More intensional versions of rice's theorem," in *Computing with Foresight and Industry - 15th Conference on Computability in Europe, CiE 2019, Durham, UK, July 15-19, 2019, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11558, Springer, 2019, pp. 217–229.

[43] P. Baldan, F. Ranzato, and L. Zhang, "A rice's theorem for abstract semantics," in *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, ser. LIPIcs, vol. 198, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 117:1–117:19.

[44] R. Giacobazzi, F. Logozzo, and F. Ranzato, "Analyzing program analyses," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, 2015, pp. 261–273.

[45] P. Cousot and R. Cousot, "Formal language, grammar and set-constraint-based program analysis by abstract interpretation," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, ACM, 1995, pp. 170–181.

[46] A. Aiken, "Introduction to set constraint-based program analysis," *Sci. Comput. Program.*, vol. 35, no. 2, pp. 79–111, 1999.

[47] T. W. Reps, "Program analysis via graph reachability," *Inf. Softw. Technol.*, vol. 40, no. 11-12, pp. 701–726, 1998.

[48] V. Vazirani, *Approximation Algorithms*. Springer Berlin Heidelberg, 2013, ISBN: 9783662045657.

[49] S. Arora, "The approximability of np-hard problems," in *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, ACM, 1998, pp. 337–348.

[50] M. Blum, "A machine-independent theory of the complexity of recursive functions," *J. ACM*, vol. 14, no. 2, pp. 322–336, 1967.

[51] A. Asperti, "The intensional content of rice's theorem," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, 2008, pp. 113–119.

[52] H. Rogers, "Gödel numberings of partial recursive functions," *The journal of symbolic logic*, vol. 23, no. 3, pp. 331–341, 1958.

[53] C. Papadimitriou, *Computational Complexity* (Theoretical computer science). Addison-Wesley, 1994, ISBN: 9780201530827.

[54] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, fourth edition*. MIT Press, 2022, ISBN: 9780262367509.

[55] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, ACM, 1971, pp. 151–158.

[56] L. A. Levin, "Universal sequential search problems," *Problemy peredachi informatsii*, vol. 9, no. 3, pp. 115–116, 1973.

[57] S. A. Cook, "Short propositional formulas represent nondeterministic computations," *Inf. Process. Lett.*, vol. 26, no. 5, pp. 269–270, 1988.

[58] S. Aaronson, "P =? NP," in *Open Problems in Mathematics*, Springer, 2016, pp. 1–122.

[59] A. Kolokolova, "Complexity barriers as independence," *The Incomputable: Journeys Beyond the Turing Barrier*, pp. 143–168, 2017.

[60] S. Aaronson and A. Wigderson, "Algebrization: A new barrier in complexity theory," *ACM Trans. Comput. Theory*, vol. 1, no. 1, 2:1–2:54, 2009.

[61] L. Fortnow, "Diagonalization," *Bull. EATCS*, vol. 71, pp. 102–113, 2000.

[62] A. Nash, R. Impagliazzo, and J. Remmel, "Universal languages and the power of diagonalization," in *18th IEEE Annual Conference on Computational Complexity, 2003. Proceedings.*, IEEE, 2003, pp. 337–346.

[63] S. Ding and Q. Zhang, "Witnessability of undecidable problems," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 982–1002, 2023.

[64] P. Pratikakis, J. S. Foster, and M. Hicks, "Existential label flow inference via CFL reachability," in *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4134, Springer, 2006, pp. 88–106.

[65] J. Rehof and M. Fähndrich, "Type-base flow analysis: From polymorphic subtyping to cfl-reachability," in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, ACM, 2001, pp. 54–66.

[66] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, ACM Press, 1995, pp. 49–61.

[67] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with cfl-reachability," in *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7791, Springer, 2013, pp. 61–81.

[68] Y. Su, D. Ye, and J. Xue, "Parallel pointer analysis with cfl-reachability," in *43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014*, IEEE Computer Society, 2014, pp. 451–460.

[69] J. Kodumal and A. Aiken, "The set constraint/cfl reachability connection in practice," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, ACM, 2004, pp. 207–218.

[70] T. W. Reps, "Program analysis via graph reachability," *Inf. Softw. Technol.*, vol. 40, no. 11-12, pp. 701–726, 1998.

[71] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990*, ACM Press, 1990, pp. 230–242.

[72] D. Melski and T. W. Reps, "Interconvertibility of a class of set constraints and context-free-language reachability," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 29–98, 2000.

[73] S. Chaudhuri, "Subcubic algorithms for recursive state machines," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, 2008, pp. 159–169.

[74] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast algorithms for dyck-cfl-reachability with applications to alias analysis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, ACM, 2013, pp. 435–446.

[75] K. Chatterjee, B. Choudhary, and A. Pavlogiannis, "Optimal dyck reachability for data-dependence and alias analysis," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 30:1–30:30, 2018.

[76] Y. Lei, Y. Sui, S. Ding, and Q. Zhang, "Taming transitive redundancy for context-free language reachability," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 1556–1582, 2022.

[77] Q. Zhang and Z. Su, "Context-sensitive data-dependence analysis via linear conjunctive language reachability," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, ACM, 2017, pp. 344–358.

[78] Y. Li, Q. Zhang, and T. W. Reps, "Fast graph simplification for interleaved dyck-reachability," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, ACM, 2020, pp. 780–793.

[79] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, ACM, 2006, pp. 387–400.

[80] D. Yan, G. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for java," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, ACM, 2011, pp. 155–165.

[81] V. Kahlon, "Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks," in *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, IEEE Computer Society, 2009, pp. 27–36.

[82] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[83] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 48:1–48:29, 2019.

[84] M. A. Harrison, *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc., 1978.

[85] S. C. Kleene, "Introduction to metamathematics," 1952.

[86] G. A. Kildall, "A unified approach to global program optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, ACM Press, 1973, pp. 194–206.

[87] P. Cousot, "Asychronous iterative methods for solving a fixed point system of monotone equations in a complete lattice.," 1977.

[88] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, ACM, 2015, pp. 106–117.

[89] SPEC, *Spec cpu 2017*, https://www.spec.org/cpu2017/, Accessed: Nov 6, 2022, 2017.

[90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[91] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, 1986.

[92] A. H. Kjelstrøm and A. Pavlogiannis, "The decidability and complexity of interleaved bidirected dyck reachability," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–26, 2022.

[93] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, IEEE Computer Society, 2004, pp. 75–88.

[94] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, ACM, 2016, pp. 265–266.

[95] X. Xiao, Q. Zhang, J. Zhou, and C. Zhang, "Persistent pointer information," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, ACM, 2014, pp. 463–474.

[96] A. Milanova, "Flowcfl: Generalized type-based reachability analysis: Graph reduction and equivalence of cfl-based and type-based reachability," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 178:1–178:29, 2020.

[97] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, ACM, 1998, pp. 85–96.

[98] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[99] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, 9:1–9:34, 2008.

[100] W. Anggoro and J. Torjo, *Boost. Asio C++ Network Programming*. Packt Publishing Ltd, 2015.

[101] L. Z. Eng, *Qt5 C++ GUI programming cookbook*. Packt Publishing Ltd, 2016.

[102] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with tensorflow: A review," *Journal of Educational and Behavioral Statistics*, vol. 45, no. 2, pp. 227–248, 2020.

[103] N. M. Josuttis, "The c++ standard library: A tutorial and reference," 2012.

[104] Cppreference, *Sfinae*, https://en.cppreference.com/w/cpp/language/sfinae, Accessed in April 2024.

[105] Tumblr, *The grand cpp error explosion competition*, https://tgceec.tumblr.com/, Accessed in April 2024.

[106] StackExchange, *Generate the longest error message in cpp*, https://codegolf.stackexchange.com/a/10470, Accessed in April 2024.

[107] StackOverflow, *Deciphering cpp template error messages*, https://stackoverflow.com/questions/47980/deciphering-c-template-error-messages, Accessed in April 2024.

[108] StackOverflow, *How to improve compiler error messages when using cpp std::visit?* https://stackoverflow.com/questions/72507596/how-to-improve-compiler-error-messages-when-using-c-stdvisit, Accessed in April 2024.

[109] G. D. Reis and B. Stroustrup, "Specifying C++ concepts," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, ACM, 2006, pp. 295–308.

[110] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3586, Springer, 2005, pp. 428–452.

[111] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, ACM, 2016, pp. 607–618.

[112] Y. Peng *et al.*, "Static inference meets deep learning: A hybrid type inference approach for python," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, pp. 2019–2030.

[113] B. Stroustrup, *A Tour of C++*. Addison-Wesley Professional, 2022.

[114] W. Gasarch, "A survey of recursive combinatorics," in *Studies in Logic and the Foundations of Mathematics*, vol. 139, Elsevier, 1998, pp. 1041–1176.

[115] ISO, *The standard definition of cpp*, https://www.iso.org/standard/79358.html, Accessed in October 2024.

[116] R. Garcia and A. Lumsdaine, "Toward foundations for type-reflective metaprogramming," in *Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings*, ACM, 2009, pp. 25–34.

[117] W. V. Quine, "The problem of simplifying truth functions," *The American mathematical monthly*, vol. 59, no. 8, pp. 521–531, 1952.

[118] W. V. Quine, "A way to simplify truth functions," *The American mathematical monthly*, vol. 62, no. 9, pp. 627–631, 1955.

[119] E. J. McCluskey, "Minimization of boolean functions," *The Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[120] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.

[121] B. Livshits *et al.*, "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[122] Cppreference, *Template argument deduction*, https://en.cppreference.com/w/cpp/language/template_argument_deduction, Accessed in April 2024.

[123] Cppreference, *Class template argument deduction*, https://en.cppreference.com/w/cpp/language/class_template_argument_deduction, Accessed in April 2024.

[124] Cppreference, *Named requirements*, https://en.cppreference.com/w/cpp/named_req, Accessed in April 2024.

[125] Cppreference, *Cpp named requirements: Legacyinputiterator*, https://en.cppreference.com/w/cpp/named_req/InputIterator, Accessed in April 2024.

[126] StackOverflow, *Compilation error for a binary search on the attributes*, https://stackoverflow.com/questions/42604796/compilation-error-for-a-binary-search-on-the-attributes, Accessed in April 2024.

[127] StackOverflow, *Using stdsort in cpp with iterators and templates*, https://stackoverflow.com/questions/59096297/using-stdsort-in-c-with-iterators-and-templates, Accessed in October 2024.

[128] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[129] LLVM, *Llvm language reference manual*, https://llvm.org/docs/LangRef.html, Accessed in April 2024.

[130] J. Altidor, S. S. Huang, and Y. Smaragdakis, "Taming the wildcards: Combining definition- and use-site variance," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, ACM, 2011, pp. 602–613.

[131] J. Altidor and Y. Smaragdakis, "Refactoring java generics by inferring wildcards, in practice," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, ACM, 2014, pp. 271–290.

[132] G. Li, I. Ghosh, and S. P. Rajan, "KLOVER: A symbolic execution and automatic test generation tool for C++ programs," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 609–615.

[133] G. Li, I. Ghosh, and S. P. Rajan, "Kil: An abstract intermediate language for symbolic execution and test generation of c++ programs," Citeseer, 2012, p. 15.

[134] J. Barnat *et al.*, "Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 863–868.

[135] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for C/C++ concurrency," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 17:1–17:32, 2018.

[136] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022.

[137] B. Norris and B. Demsky, "A practical approach for model checking C/C++11 code," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, 10:1–10:51, 2016.

[138] J. G. Siek and W. Taha, "A semantic analysis of C++ templates," in *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 304–327.

[139] J. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. P. Priesnitz, "A comparison of c++ concepts and haskell type classes," in *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2008, Victoria, BC, Canada, September 20, 2008*, ACM, 2008, pp. 37–48.

[140] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, 7:1–7:50, 2019.

[141] C. Wang, P. Yao, W. Tang, Q. Shi, and C. Zhang, "Complexity-guided container replacement synthesis," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, pp. 1–31, 2022.

[142] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, ACM, 2007, pp. 425–434.

[143] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational API inference," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, ACM, 2022, pp. 44–56.

[144] S. Xu, Z. Dong, and N. Meng, "Meditor: Inference and application of API migration edits," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, IEEE / ACM, 2019, pp. 335–346.

[145] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, ACM, 2007, pp. 123–134.

[146] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, ACM, 2009, pp. 75–86.

[147] V. Chibotaru, B. Bichsel, V. Raychev, and M. T. Vechev, "Scalable taint specification inference with big code," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, ACM, 2019, pp. 760–774.

[148] T. W. Reps, "Undecidability of context-sensitive data-independence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, 2000.

[149] Q. Zhang and Z. Su, "Context-sensitive data-dependence analysis via linear conjunctive language reachability," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, 2017, pp. 344–358.

[150] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[151] M. Jeon and H. Oh, "Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–29, 2022.

[152] S. Ding and Q. Zhang, "Mutual refinements of context-free language reachability," in *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portu-*

gal, October 22-24, 2023, Proceedings, ser. Lecture Notes in Computer Science, vol. 14284, Springer, 2023, pp. 231–258.

[153] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, 2017, pp. 599–612.

[154] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017.

[155] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, ACM, 2010, pp. 313–326.

[156] L. D'Antoni, Q. Hu, J. Kim, and T. W. Reps, "Programmable program synthesis," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 12759, Springer, 2021, pp. 84–109.

[157] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[158] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, pp. 2130–2141.

[159] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, 2012, pp. 335–346.