View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Z3str4: A Solver for Theories over Strings

by

Murphy Berzish

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Engineering

Waterloo, Ontario, Canada, 2021

© Murphy Berzish 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Loris D'Antoni
Assistant Professor, Dept. of Computer Sciences
University of Wisconsin

Supervisor: Vijay Ganesh
Associate Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

Internal Members: Krzysztof Czarnecki
Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

Derek Rayside
Associate Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

Internal-External Member: Joanne Atlee
Professor, Cheriton School of Computer Science
University of Waterloo

Author's Declaration

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

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

Abstract

Satisfiability Modulo Theories (SMT) solvers supporting rich theories of strings have facilitated numerous industrial applications with the need to reason about string operations and predicates that are present in many popular programming languages. Constraints encountered in practical applications have immense value in inspiring new algorithms and heuristics that string solvers can take advantage of to tackle new, more difficult problems. This is especially relevant as the combinations of operators typically supported by string solvers, or that are encountered in program analysis constraints, quickly result in theories whose satisfiability problems are undecidable.

I present a number of theoretical and practical contributions in the domain of string solving. On the theoretical side, I illustrate decidability and undecidability results related to different relevant theories which include strings. On the practical side, I describe a collection of algorithms and heuristics designed to address challenges encountered in applications of string solvers, culminating with the introduction of Z3str4, a state-of-the-art solver for theories over strings. Z3str4 incorporates many improvements over its predecessor Z3str3, including an algorithm selection architecture that takes advantage of multiple solving algorithms in order to leverage the strengths of diverse string solving procedures against formulas they are predicted to be able to solve efficiently. I also present a back-end model construction algorithm for Z3str4 which is a hybrid between word-based and unfolding-based algorithms. Furthermore, I showcase the power of Z3str4 against other state-of-the-art tools in an empirical evaluation over a large and diverse collection of benchmarks. Additionally, I describe algorithms and heuristics specific to solving regular expression constraints, and demonstrate their effectiveness in a detailed and focused empirical evaluation.

Acknowledgements

This thesis and the work I discuss here would not have been possible without the support, guidance, and encouragement of my supervisor Dr. Vijay Ganesh. I am tremendously thankful for his suggestions, ideas, and feedback in all of the work I have presented.

I am deeply grateful to my collaborator Mitja Kulczynski for his tireless efforts in contributing to Z3str4 and providing computing power and support to run the experiments I present here, and for his valuable feedback on this thesis.

I am also deeply grateful to my collaborator Federico Mora for his contributions to Z3str4 and the numerous ideas and suggestions he has made to improve the tool, and for his valuable feedback on this thesis.

I am very happy to have had the opportunity to work with an amazing Ph.D committee at the University of Waterloo in Drs. Joanne Atlee, Krzysztof Czarnecki, and Derek Rayside, and would especially like to thank my external committee member Dr. Loris D'Antoni for participating in my thesis defence.

I am very grateful to Florin Manea, Dirk Nowotka, and Joel Day for their coauthorship and collaboration on previous papers related to Z3str4.

Finally, I am thankful for the feedback and suggestions provided by the many anonymous readers and reviewers of the submitted and accepted papers upon which my work was based, whose comments have assisted me in preparing stronger papers.

Dedication

This thesis is dedicated to my parents.

Table of Contents

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 Applications of String Solvers . . . 2

1.2 Related Work . . . 3

1.2.1 Theory . . . 3

1.2.2 Practice . . . 4

1.3 Contributions . . . 5

2 Background 7

2.1 Syntax of Z3str4's Input Language . . . 7

2.2 Definitions and Semantics . . . 7

2.2.1 Satisfiability, Complexity, and Decidability . . . 10

3 (Un)decidability Results for Theories over Strings 12

3.1 Undecidability of the Theory of Strings with String-Number Conversion . . 13

3.1.1 The Theory of Power Arithmetic T_p and Büchi's Results . . . 13

3.1.2 Proof of Undecidability of $T_{L,n,c}$ . . . 13

3.1.3 Expressibility of $\pi(x, y, z)$ . . . 15
3.2 Decidability of the Theory of Strings with Length and Regular Expression Constraints . . . 17

4 Z3str4: A Solver for Theories Over Strings **21**
4.1 Motivation . . . 21
4.2 Architecture of Z3str4 . . . 22
4.3 Algorithm Selection . . . 23
4.4 Length Abstraction Solver . . . 27
4.4.1 MultisetCheck Subroutine . . . 29
4.5 Fixed-Length Model Construction . . . 29
4.5.1 Solving Strings via Arrangements . . . 31
4.5.2 A Bit-Vector Backend for Solving String Constraints . . . 33
4.6 Theory-Aware Heuristics . . . 36
4.6.1 Theory-Aware Branching . . . 37
4.6.2 Theory-Aware Case Split . . . 38
4.7 Clause Sharing . . . 40
4.8 Empirical Evaluation . . . 41
4.8.1 Empirical Setup and Solvers Used . . . 41
4.8.2 Benchmarks Used . . . 42
4.8.3 Results and Analysis . . . 44
4.8.4 Performance Analysis of Components of Z3str4 . . . 44

5 Algorithms and Heuristics for Theories over Regular Expressions and Linear Arithmetic over String Length **53**
5.1 Background and Motivation . . . 53
5.2 Algorithm for Solving Regex, Length, and Linear Arithmetic Constraints . 56
5.3 Length-Aware Heuristics for Solving Regular Expression Constraints 58
5.3.1 Computing Length Information from Regexes . . . 59

5.3.2 Optimizing Automata Operations via Length Information 59
5.3.3 Leveraging Length Information to Optimize Search 60
5.3.4 Constructing Over-Approximated Prefixes/Suffixes to Find Empty Intersections 60
5.4 Empirical Evaluation 62
5.4.1 Empirical Setup and Solvers Used 62
5.4.2 Benchmarks 63
5.4.3 Comparison and Scoring Methods 68
5.4.4 Analysis of Empirical Results 69
5.4.5 Detailed Experimental Results 71
5.4.6 Analysis of Individual Heuristics and Results 71

6 Observations and Future Work **73**
6.1 Future Work 73
6.1.1 Theoretical Results 73
6.1.2 Algorithm Selection 74
6.1.3 Solving Regular Expression Constraints 74
6.1.4 Theory-Aware Heuristics 75
6.1.5 Applications of String Solvers 75
6.2 Conclusion 76

References **77**

List of Figures

1.1 How an SMT solver can be used in program analysis applications. 3

2.1 Syntax of input formulas accepted by Z3str4. 8

3.1 General form of length automaton with stem length s and period length p. All transitions are on the symbol '1'. 18

4.1 Architecture of the Z3str4 tool. 23

4.2 The three-tiered probe used by Z3str4 to perform arm selection. 26

4.3 Possible arrangements of $A \cdot B = X \cdot Y$. 32

4.4 Illustration of an overlapping variable X in $0 \cdot X = X \cdot 0$. 32

4.5 Architecture of Z3str4's fixed-length model construction algorithm. 33

4.6 Cactus plot showing performance of string solvers on combined benchmarks. 43

4.7 All three component solvers on queries in the conjunctive fragment. 46

4.8 All three component solvers on queries outside the conjunctive fragment. . 48

4.9 Cactus plot of string solvers on all benchmarks. Timeout=20 s. Timeout, unknown, and error instances excluded. 48

4.10 Performance of all three component solvers on queries in the "few word equations" fragment 50

4.11 Performance of all three component solvers on queries in the "many word equations" fragment 51

4.12 Cactus plot of Z3str4 on all benchmarks with clause sharing enabled/disabled. Timeout=20 s. Timeout, unknown, and error instances excluded. 51

5.1 Architecture of Z3str4's regular expression solving algorithm. 58
5.2 Cactus plot summarizing detailed performance on Automatark benchmark. 65
5.3 Cactus plot showing detailed results for the StringFuzz-regex-generated benchmark. 66
5.4 Cactus plot showing detailed results for the StringFuzz-regex-transformed benchmark. 67
5.5 Cactus plot showing detailed performance for the RegEx-collected benchmark. 68
5.6 Cactus plot summarizing performance on all benchmarks. Z3str4 has the best overall performance. 69
5.7 Cactus plot comparing performance by disabling individual heuristics on all benchmarks. 72

List of Tables

4.1 Table showing detailed results of string solvers on combined benchmarks. . 43

4.2 All three component solvers on queries in the conjunctive fragment. 45

4.3 All three component solvers on queries outside the conjunctive fragment. . 47

4.4 Cumulative Results. Timeout=20 s. Total time includes all solved, timeout, unknown and error instances. 49

4.5 All three component solvers on queries in the "few word equations" fragment. 49

4.6 Performance of all three component solvers on queries in the "many word equations" fragment. 50

4.7 Cumulative results for Z3str4 with clause sharing enabled/disabled. Timeout=20 s. Total time includes all solved, timeout, unknown and error instances. 52

5.1 Detailed results for the Automatark benchmark. Z3str4 has the biggest lead with a score of 1.01. 65

5.2 Detailed results for the StringFuzz-regex-generated benchmark. Z3str4 has the biggest lead with a score of 1.25. 66

5.3 Detailed results for the StringFuzz-regex-transformed benchmark. Z3str4 has the biggest lead with a score of 1.0. 67

5.4 Detailed results for the RegEx-collected benchmark. CVC4 has the biggest lead with a score of 1.03. 68

5.5 Combined results of string solvers on all benchmarks. **Z3str4** has the best overall performance on all benchmarks compared to CVC4, OSTRICH, Z3seq, Z3str3, and Z3-trau and the biggest lead with a score of 1.02. 70

5.6 Comparison of different regular expression heuristics in Z3str4 on all benchmarks. 72

Chapter 1

Introduction

In this chapter, I explain the motivation for studying the theoretical and practical aspects of reasoning about strings, describe some related work in the field, and outline the contributions that embody the remainder of the thesis.

1.1 Motivation

Support for strings in automated theorem provers for satisfiability modulo theories (SMT) has enabled numerous applications in the context of program analysis, automated reasoning, verification, and security. Common to all of these applications is the requirement for an automated reasoning procedure supporting a rich first-order theory over strings, integer arithmetic, string length, regular expressions, and string terms and predicates such as `substr`, `contains`, and `indexof` which encode string manipulations in many popular programming languages. This, in turn, is complicated by the fact that many theories including strings that are relevant to program analysis are undecidable, such as the theory of strings with string-number conversion (Chapter 3). Additionally, reasoning about quantifier-free word equations alone is in nondeterministic LINSPACE [37, 38], and many elementary operations over automata, which can be used to reason about regular expression membership constraints, are PSPACE-complete. Thus, the task of creating efficient automated reasoning tools handling theories over strings, or less formally "string solvers", remains a very difficult challenge.

Modern string solvers implement a diverse collection of algorithms and heuristics for reasoning about strings. Conversely, new industrial applications continue to challenge the

performance of these algorithms and open the potential to introduce new heuristics and advanced algorithms for handling constraints that are practically relevant.

Solving arbitrary string instances is, of course, not tractable. However, industrial applications of string solvers typically generate and verify constraints that follow certain patterns, use certain operators in combination, and have particular structure corresponding to the task they perform. This motivates the development of string solving algorithms and heuristics that target the types of constraints commonly encountered in industrial instances, while still being able to solve as broad a set of instances as possible. In contrast with random instances, industrial string instances often contain a wealth of implicit information that can be used to guide the search and prune the search space in order to find a solution more efficiently, and algorithms and heuristics that take advantage of this information have the potential to be highly effective. This was demonstrated by the improvements of the Z3str2 string solver over its predecessor, Z3-str [72]. Both string solvers use the same fundamental search algorithm to reason about strings. The key difference is that Z3str2 leverages length information from the arithmetic solver during the search to prune unsatisfiable branches in the string solver, and asserts additional facts about the lengths of strings to the the arithmetic solver in order to guide the search in the integer domain. These heuristics enabled Z3str2 to achieve a 14x speedup in terms of total solving time over Z3-str, according to the empirical evaluation done in the same paper. Thus, the introduction of carefully designed, broadly applicable algorithms and heuristics which enable tools to take advantage of implicit information in industrial instances is of immense value in improving the capabilities of string solvers. Furthermore, the diversity of string solving algorithms can be leveraged from the point of view of an algorithm selection approach in order to leverage different algorithms based on properties of the input.

In this thesis, I advance the state of the art of string solving with both theoretical and practical contributions, culminating in the introduction of Z3str4, a string solver for a standardized theory of strings that is relevant to industrial applications. Z3str4 incorporates a multitude of new algorithms and heuristics that are directly motivated by industrial instances and practical challenges encountered when solving strings. As I will demonstrate, its performance improves upon that of several state-of-the-art string solvers, and it can solve more instances in less time than other competing tools while maintaining correctness and stability – both of which are necessities for industrial applications of such tools.

1.1.1 Applications of String Solvers

String solvers (and, more broadly, SMT solvers in general) are typically integrated as a back-end component in a larger architecture involving a program analysis or automated

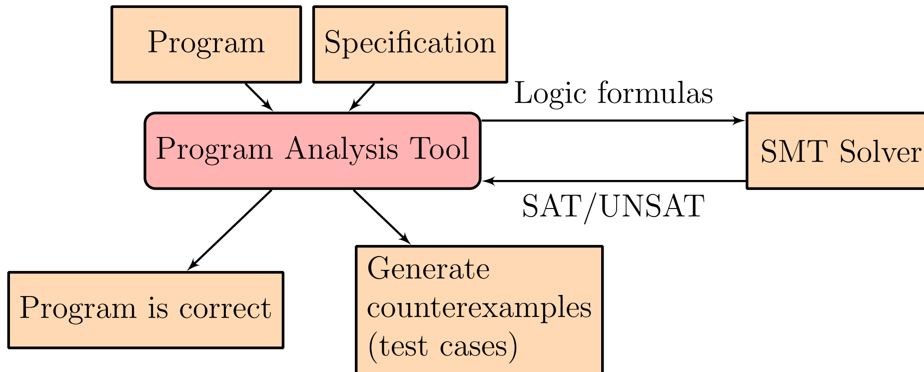

Figure 1.1: How an SMT solver can be used in program analysis applications.

reasoning tool, as illustrated in Figure 1.1. Input to the tool typically consists of a program and a specification. The program analysis tool then generates logical formulas based on the input. These formulas may correspond to program paths to be explored, input cases to be checked, etc. The SMT solver is then asked to solve these formulas and answer SAT or UNSAT for each one. Based on the answer, the tool may generate additional queries to the solver. Eventually, the tool can either conclude that the program is correct with respect to the given specification, or it can output one or more counterexamples in which the specification was not met.

The Kaluza symbolic execution engine [59] and Jalangi analysis framework [61] both use string solvers to encode string terms and predicates that are encountered during analysis of JavaScript programs for web application vulnerability scanning. IBM's PISA [65] and AppScan [1] tools use string solvers to perform security analysis of application binaries and web applications respectively. Amazon Web Services uses string solvers to reason about security policies for cloud applications as part of the AWS Zelkova service [10]. String solvers have also been used in many other symbolic execution and verification tools [16, 57].

1.2 Related Work

1.2.1 Theory

Makanin showed in 1977 that the theory of quantifier-free word equations was decidable [48]. Plandowski later showed that this problem was in PSPACE [54, 55], and Jeż later expanded this result to show that it is in nondeterministic LINSPACE [37, 38]. Schulz [60]

extended Makanin's algorithm to word equations with regex constraints. Continuing the aforementioned thread of research, Ganesh et al. proved that satisfiability for quantified word equations with a single quantifier alternation is undecidable [33]. There exist many extensions to the theory of word equations whose (un)decidability has been established [48, 55, 37, 25, 47, 32]. However, the status of the quantifier-free theory of word equations with integer length remains an open problem, and has been unsettled for several decades [48, 55, 33, 49].

1.2.2 Practice

In a recent survey paper, Amadini [8] classifies approaches to string constraint solving into three main categories: automata-based, word-based, and unfolding-based approaches. Automata-based approaches use finite automata to represent string variables and constraints. Word-based approaches reason about systems of word equations directly. Unfolding-based approaches expand string variables over the sequences of characters they represent. All three approaches have different strengths and weaknesses. The solver I present in this thesis, Z3str4, uses a combination of all three approaches identified by Amadini, and as I describe in Chapters 4 and 5, such a hybrid approach may be better than any single one.

The foundational HAMPI solver [41] used an unfolding-based approach to reduce fixed-length string constraints to bit-vectors. However, HAMPI does not support unbounded string variables. Many tools that followed HAMPI use different strategies to support unbounded string variables.

The Z3 theorem prover [27] is a DPLL(T) word-based SMT solver for theory combinations over first-order logic. Z3 includes an arithmetic solver for linear integer arithmetic, and a sequence solver that supports word-based reasoning over strings. Z3str4 is built on top of Z3, and Z3's sequence solver is included in algorithm selection. Z3str3 [15] is based on Z3 and Z3str2 [72]. It is also word-based and uses a reduction known as "arrangements". I explain this reduction in more detail in Section 4.5 as it is also part of Z3str4. Z3str4 is the latest solver in the Z3-str family, and the successor to Z3str3.

The CVC4 solver [44] handles constraints over the theory of strings and arithmetic using a word-based approach, and uses a similar DPLL(T) architecture to Z3. Norn [6] is an automata-based solver that solves integer arithmetic constraints using finite automata and then represents word equations with finite automata that have been restricted with respect to concrete length constraints. Stranger [71] is another automata-based approach, but based on a static analysis technique that determines possible solutions of a string variable

while traversing an automaton. [21] implemented another technique using transducers to solving string constraints in their tool called Ostrich. Ostrich implements a reduction from straight-line string constraints [1] to the emptiness problem of alternating finite automata. The Trau solver [4] looks for simple patterns inside the input formula within a CEGAR framework. A newer version of Trau, called Z3-Trau [3], was also implemented within the Z3 theorem prover and includes several enhancements such as a more efficient handling of string-number conversion.

1.3 Contributions

I present the following theoretical and practical contributions related to solving strings:

Contribution 1: (Un)decidability results for theories over strings. In Chapter 3, I establish a number of theoretical results related to decidability and undecidability of various theories over strings. In particular, I show that the quantifier-free first-order theory of strings with word equations, concatenation, string length, linear integer arithmetic, and string-number conversion is undecidable, and I demonstrate the decidability of the theory of strings with length and regular expression constraints via an elementary automata-based decision procedure. Establishing such results is relevant not only in advancing the theoretical understanding of string logics, but also in motivating the design of algorithms and heuristics used in string solvers. For example, the decidability result directly informs the design of the length-aware regular expression algorithm I present in Chapter 5. Conversely, understanding that a relevant theory combination is undecidable in turn inspires new heuristics geared towards solving instances that occur in practice. Additionally, establishing the border of decidability and undecidability brings us closer to addressing long-standing open problems, such as whether the theory of strings with integer length alone is decidable – a question that, as previously mentioned, has remained open for many years.

Contribution 2: Z3str4, a solver for theories over strings. In Chapter 4, I present Z3str4, an SMT solver for the theory of strings, built on top of the open-source Z3 theorem prover [27] by Microsoft Research. Z3str4 contains new architectural improvements and algorithmic enhancements that enable it to achieve performance competitive with existing state-of-the-art tools.

[1] The "straight-line" restriction is similar in principle to the concept of "solved form" introduced by Z3-str [74] and explained in Section 4.5.

I introduce a novel three-tiered algorithm selection procedure that enables Z3str4 to leverage the strengths of different string solving algorithms against instances where they are likely to perform very well. This enables Z3str4 to achieve greater performance than could be obtained by using any of these algorithms alone.

I describe a major extension which I implemented in Z3str4's arrangement solver in order to improve the process of model construction. Efficiency in this procedure is paramount to solving practical string instances, as it is always called at least once when solving any satisfiable formula, and often several times. The algorithm I describe is a combination of both word-based and unfolding-based approaches that achieves the benefits of both individual approaches without the major limitations of either.

I outline two so-called "theory-aware" heuristics which allow the string theory solver in Z3str4 to influence the branching decisions made by Z3's core solver. Allowing theory solvers to provide more information to the core solver in a CDCL(T)-style SMT solver is an interesting and natural extension of the standard architecture and potentially allows the core solver to make better branching decisions or to avoid combinations of Boolean literals that are infeasible under theory semantics without needing extra clauses to block those assignments.

To conclude this chapter, I present the results of a detailed empirical evaluation against these tools over a large and diverse benchmark of industrial, hand-crafted, and randomly-generated string instances. To the best of my knowledge, this is the largest and most comprehensive collection of string instances that has been used for testing string solvers to date.

Contribution 3: Algorithms and heuristics for theories over regular expressions and linear arithmetic over string length. In Chapter 5, I focus on algorithms and heuristics related to a particular facet of string solving: regular expression constraints. Regex constraints are very common when reasoning about formulas that describe sanitizers or input validation procedures. I illustrate some of the challenges related to reasoning about regex constraints efficiently, and describe an algorithm and several heuristics designed to address these challenges in practical settings. I also present the results of an empirical evaluation specifically targeting regex solving and compare Z3str4 to other state-of-the-art tools on a regex-heavy benchmark in order to showcase the power of this algorithm and these heuristics.

Chapter 2

Background

2.1 Syntax of Z3str4's Input Language

Z3str4 accepts input in the SMT-LIB format [14] following the currently published standard for the theory of strings. It can handle quantifier-free formulas over Boolean combinations of string, integer, and regular expression formulas and terms. Atomic formulas handled by the string solver include string equalities and disequalities, regular expression membership constraints, and extended string predicates such as `contains`, `prefixof`, `suffixof`, etc. Atomic formulas over integers, which may include inequalities, are handled by Z3's arithmetic solver. Z3str4 supports additional integer-sorted terms that integrate with strings, such as `str.len` and `str.to_int`. Boolean combinations of atomic formulas are handled by Z3's core solver in conjunction with the string and arithmetic solvers in a DPLL(T)-style approach. A summary of the basic syntax of Z3str4's input language is presented in Figure 2.1. Con_{str} and $Con_{\mathbb{Z}}$ denote the set of all string constants and integer constants, respectively, over a fixed-length alphabet (practically, ASCII or Unicode). Var_{str} and Var_{int} denote the infinite sets of all string variables and integer variables, respectively.

2.2 Definitions and Semantics

In this section, I outline a number of common definitions used in the results to follow, and define the semantics of terms in the input language and theories considered in this thesis and in accordance with the SMT-LIB standard for strings [14].

$$
\begin{array}{lcl}
F & ::= & Atom \;\mid\; F \wedge F \;\mid\; F \vee F \;\mid\; \neg F \\
Atom & ::= & t_{str} = t_{str} \;\mid\; t_{str} \neq t_{str} \;\mid\; A_{int} \;\mid \\
 & & A_{ext} \;\mid\; A_{re} \\
A_{re} & ::= & t_{str} \in t_{re} \\
A_{int} & ::= & t_{int} = t_{int} \;\mid\; t_{int} < t_{int} \\
A_{ext} & ::= & \text{str.contains}(t_{str}, t_{str}) \;\mid\; \text{str.prefixof}(t_{str}, t_{str}) \;\mid \\
 & & \text{str.suffixof}(t_{str}, t_{str}) \\
t_{int} & ::= & m \mid v \mid \text{len}(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int} \;\mid \\
 & & \text{str.indexof}(t_{str}, t_{str}, t_{int}) \;\mid\; \text{str.to_int}(t_{str}) \\
 & & \text{where } m \in Con_{\mathbb{Z}} \;,\; v \in Var_{int} \\
t_{str} & ::= & s \;\mid\; v \;\mid\; t_{str} \cdot t_{str} \;\mid \\
 & & \text{str.from_int}(t_{int}) \;\mid\; \text{str.replace}(t_{str}, t_{str}, t_{str}) \;\mid \\
 & & \text{str.at}(t_{str}, t_{int}) \;\mid\; \text{str.substr}(t_{str}, t_{int}, t_{int}) \\
 & & \text{where } s \in Con_{str} \;,\; v \in Var_{str} \\
t_{re} & ::= & \text{``}Con_{str}\text{''} \;\mid\; t_{re} \cdot t_{re} \;\mid\; t_{re} \cup t_{re} \;\mid\; t_{re}^{*} \;\mid\; t_{re}^{-}
\end{array}
$$

Figure 2.1: Syntax of input formulas accepted by Z3str4.

A **string** is an ordered sequence of zero or more elements, or **characters**, taken from a set Σ called an **alphabet**. The alphabet Σ is taken to be finite unless otherwise stated. The set of all strings over a given alphabet Σ is denoted by Σ^*. The **empty string**, denoted ϵ, is the unique string consisting of zero characters. Two strings are **equal** if they consist of the same characters appearing in the same order.

The positions of characters in a string are numbered with non-negative integers, and are zero-based. The first character of a non-empty string occurs at position 0. The `str.at` function extracts the character at position I from a string S and returns it as a string of length 1; if I is not in the range $[0, len(S))$, then the function's value is defined to be the empty string. [1]

The **length** of a string is the number of characters it contains. The empty string has length 0. The string length function, denoted len, is a function from strings to non-negative integers.

The **concatenation** of two strings A and B is another string consisting of all of the

[1] The SMT-LIB standard defines `str.at` – and many other operators – in this fashion because all functions must be total, and therefore must be well-defined for all possible values of their domain, even if those values correspond to semantics that are not ordinarily considered meaningful. As a more extreme example, integer and real division by zero are not inherently considered "errors" in those theories.

characters in A in the same order followed by all of the characters in B in the same order. Concatenation, denoted $\cdot$, is typically written as an infix operator (i.e. $A \cdot B$ is the concatenation of A followed by B). [2]

The **prefix** and **suffix** predicates `str.prefix_of` and `str.suffix_of` are defined as follows: X is a prefix (resp. suffix) of Y iff there exists a (possibly empty) string T such that $Y = X \cdot T$ (resp. $Y = T \cdot X$). Y **contains** X (`str.contains`) iff there exist (possibly empty) strings T_1, T_2 such that $Y = T_1 \cdot X \cdot T_2$.

The **index** of a string in another is defined as follows: `str.indexof`(A, B, I) is the integer position, counting from 0, of the first occurrence of the string B in the string A at or after the character at position I of A, if it exists; otherwise, it is -1.

A **substring** of a string is a continuous sequence of characters extracted from that string starting from a given position and of a given length. `str.substr`(X, I, N) is defined as the longest continuous sequence of characters of length at most N that can be taken from X starting at position I. If fewer than N characters can be taken, as many as possible are taken. In the case where N is negative or I is not in the range $[0, len(X))$, the substring is defined to be the empty string.

The **replace** function over a string is defined as follows: `str.replace`(S, T, T') is defined as the string obtained by replacing the first occurrence of T in S with T'. If T does not occur in S, then S is returned unmodified. In the special case where T is the empty string, the result is defined as $T' \cdot S$.

The **regular expression membership** predicate, written infix as $S \in R$ for string term S and regular expression R, is true if S is in the regular language described by R. Without exception, R must be grounded and cannot contain variables. The semantics of regular expression membership are defined by structural recursion as follows:

$S \in$ "w"	iff	$S = w$ (where w is a string constant)
$S \in R_1 \cdot R_2$	iff	there exist strings S_1, S_2 with $S = S_1 \cdot S_2$, $S_1 \in R_1$, $S_2 \in R_2$
$S \in R_1 \cup R_2$	iff	either $S \in R_1$ or $S \in R_2$
$S \in R^*$	iff	either $S = \epsilon$ or there exists a positive integer n such that $S = S_1 \cdot S_2 \cdot \ldots \cdot S_n$ and $S_i \in R$ for each $i = 1 \ldots n$
$S \in \overline{R}$	iff	$S \notin R$ (that is, $S \in R$ is false)

The **string to integer** conversion `str.to_int`(S) is defined as the non-negative integer denoted by S interpreted as a base-10 representation. Note that S may contain leading zeroes. If S is empty or contains non-digits, the result is defined to be -1. The **integer to string** conversion `str.from_int`(I) is defined as the string corresponding to the base-10

[2] The SMT-LIB standard uses the operator name `str.++` for concatenation.

representation of I with no leading zeroes. If I is negative, the result is defined to be the empty string.

For the purposes of illustrating several theoretical results more easily, I also define a non-standard **string-number conversion predicate** *numstr*. This predicate has the following semantics: $numstr(n, s)$ is true for a given integer n and string s iff s is a valid binary representation of the number n (possibly with leading zeros) and n is a non-negative integer, that is, s only contains the characters 0 and 1, and $\sum_{i=0}^{len(s)-1} s'[i]2^{len(s)-i-1} = n$, where $s'[i]$ is 0 if the ith character in s is '0' and 1 if that character is '1'. Please note that this predicate is defined over base-2 representations of integers, whereas the standardized string-integer conversion functions are defined over base-10 representations.

2.2.1 Satisfiability, Complexity, and Decidability

Given a formula ϕ, an **assignment** (or **model**) for ϕ with respect to an alphabet Σ is a map from the set of free variables in ϕ to $\Sigma^* \cup \mathbb{N}$ such that string variables are mapped to string constants in Σ^* and integer variables are mapped to integer constants in $\mathbb{N}$. If the assertions made by ϕ with respect to an assignment are true, then we say that ϕ is true under that assignment.

A formula ϕ is **satisfiable** iff there exists some assignment under which ϕ is true. If no such assignment exists, we say that ϕ is **unsatisfiable**. A formula ϕ is **valid** if it is true under all possible assignments. For two formulas ϕ, ψ, if it is the case that ϕ is satisfiable iff ψ is satisfiable, then we say that ϕ and ψ are **equisatisfiable**. Note that this definition is deliberately very broad; ϕ and ψ may have different numbers of satisfying assignments and need not even be from the same language.

The **satisfiability problem** for a set S of formulas is the problem of deciding whether any given formula in S is satisfiable. The satisfiability problem for S is **decidable** if there exists an algorithm (or decision procedure) that correctly decides the satisfiability of every formula in S. Conversely, a problem is **undecidable** if it can be demonstrated that no decision procedure for that problem can possibly be constructed. For example, the "halting problem" for Turing machines is known to be undecidable. Demonstrating undecidability can be done by showing how a supposed decision procedure for one problem could be used to construct a decision procedure for a problem that is already known to be undecidable.

A decision procedure for S must have three properties: **soundness**, **completeness**, and **termination**. An algorithm is sound if whenever it returns that an input formula is satisfiable, the input formula really is satisfiable. An algorithm is complete if for any

input formula that is satisfiable, the algorithm returns that it is satisfiable. An algorithm is terminating if it returns an answer in finite time for all inputs. In practice, the completeness and termination requirements may be relaxed for the sake of improved typical performance. (The soundness requirement is almost never relaxed as otherwise, practically speaking, the user would not be able to trust the answer given by the algorithm.)

Computer programs that are designed to solve the Boolean satisfiability problem are called **SAT solvers**. In this thesis, I primarily consider extensions of SAT to the domain of **Satisfiability Modulo Theories (SMT)**. The SMT problem is a decision problem for formulas of first-order logic with equality that are additionally expressed with respect to one or more "theories". These theories include, for example, the theory of integers, the theory of reals, the theory of bit-vectors, and most relevant here, the theory of strings [3]. A computer program that handles SMT formulas is called an **SMT solver**. Different SMT solvers are available and may offer support for different combinations of theories that can be used. The typical architecture of an SMT solver includes a **core solver** that handles Boolean constraint propagation, Boolean satisfiability solving, and conflict clause learning, and which interacts with one or more **theory solvers** which are responsible for checking the consistency of the Boolean abstraction of the input formula with respect to the logical theories they handle. This combination is sometimes referred to as DPLL(T) or CDCL(T), referring to the combination of DPLL/CDCL procedures for solving Boolean satisfiability with the extension over theories T.

We can further classify decidable problems into classes based on their time/space complexity. The complexity class **NP** is the set of all decision problems for which a "yes" answer can be verified in polynomial time (with respect to the size of the input) by a non-deterministic Turing machine. A decision problem is **NP-complete** if that problem is in NP and every other problem in NP is reducible to that one in polynomial time. The most fundamental NP-complete problem is the Boolean satisfiability problem, as established by the Cook-Levin theorem. A decision problem is **NP-hard** if every other problem in NP is reducible in polynomial time to that one (but the original problem need not be in NP itself). The class **PSPACE** is the set of all decision problems that can be solved by a Turing machine using a polynomial amount of space (with respect to the size of the input). It is known that NP is contained in PSPACE, thus making problems in PSPACE informally "harder than" problems in NP. A decision problem is **PSPACE-complete** if it is in PSPACE and every other problem in PSPACE is reducible to that one in polynomial time. The class **LINSPACE** is the set of all decision problems that can be solved by a Turing machine in linear space. It can be shown that NP $\neq$ LINSPACE.

[3] Not to be confused with the theoretical framework of "string theory" in physics.

Chapter 3

(Un)decidability Results for Theories over Strings

In this chapter, I present a number of decidability, undecidability, and complexity results for various theories over strings. Establishing decidability or undecidability of a logic informs the types of approaches that can be taken when solving instances of formulas in that logic which occur in practice (i.e. from applications). On the one hand, an undecidability result immediately implies that we cannot hope to have an efficient general-purpose algorithm that can handle all problems in that logic. This means that we should turn our attention to developing heuristics and special-purpose algorithms that can tackle as broad a space of problems as possible which we are most interested in solving. On the other hand, establishing decidability or showing a decision procedure that can be constructed in a new way motivates the integration of this decision procedure with SMT solvers. As a practical example, the decision procedure for length and regex constraints I derive in Section 3.2 directly establishes and shows the correctness of the practical algorithm I illustrate in Section 5.2. Finally, from a theoretical perspective, such results are interesting because they bring the state of the art closer, if even by steps, to tackling long-standing open problems such as decidability of quantifier-free word equations with length (as previously mentioned, this problem has been unresolved for decades).

3.1 Undecidability of the Theory of Strings with String-Number Conversion

In this section, I establish a number of results related to the undecidability of the quantifier-free first-order theory $T_{L,n,c}$ of word equations, concatenation, linear integer arithmetic, string length, and string-number conversion. [1]

3.1.1 The Theory of Power Arithmetic T_p and Büchi's Results

I first present a preliminary result due to Büchi [20] for a theory T_p known as "power arithmetic". The theory T_p has the structure $\langle \mathbb{N}, 0, 1, +, \pi, <_n, =_n \rangle$ where $\mathbb{N}$ is the set of natural numbers, 0 and 1 are distinct natural number constants, + is the two-operand addition function, $<_n$ and $=_n$ are the two-operand comparison and equality predicates, and π is a three-operand predicate defined as $\pi(x, y, z) \iff z = x2^y$. Note that only the quantifier-free fragment of this theory is considered (and in particular the satisfiability problem for the existential closure over quantifier-free formulas of T_p).

With these definitions it is possible to present the necessary context for Büchi's undecidability result for T_p. Lemmas 1 and 2, as well as the statement of Theorem 3, are adapted from [20] where they were originally presented.

Lemma 1. (Julia Robinson's divisibility lemma) *If $m \leq n, l > 2n^2$, and $l + m, l - m | l^2 - n$, then $m^2 = n$. (Refer to Lemma 5 in [20].)*

Lemma 2. (Büchi's Lemma) *In $T_p = \langle \mathbb{N}, 0, 1, +, \pi \rangle$ it is possible to existentially define addition and multiplication on $\mathbb{N}$. (Refer to Lemma 6 in [20].)*

Theorem 3. (Büchi's Undecidability Theorem) *The existential theory of $T_p = \langle \mathbb{N}, 0, 1, +, \pi \rangle$ is undecidable. (Corollary 5 in [20].)*

3.1.2 Proof of Undecidability of $T_{L,n,c}$

Recall that the satisfiability problem for the theory of quantifier free string equations with string length remains open. Knowing whether that theory is decidable would be of value in many program analysis applications. The theory $T_{L,n,c}$ I consider here is also inspired

[1] These results were obtained jointly with Vijay Ganesh, Florin Manea, and Joel Day.

by program analysis, and is a simple extension that covers an operation that is very commonly used in programming languages. The string-numeric conversion predicate models common API functions such as JavaScript's `parseInt` and `toNumber` methods which perform integer-string and string-integer conversion. Supporting string concatenation, string comparison/assignment (via equality), and conversion to and from integers represents a minimal but expressive and useful theory for practical applications. However, as I will demonstrate, even this simple extension is enough to establish undecidability.

The structure of $T_{L,n,c}$ is $\langle \Sigma^*, \mathbb{N}, 0_s, 1_s, \cdot, 0_n, 1_n, +, len, numstr, =_s, =_n, <_n \rangle$ where Σ^* is the set of all string constants, $\mathbb{N}$ is the set of all natural numbers, $\cdot$ is the string concatenation function, $+$ is the two-operand addition function for natural numbers, *len* is the string length function, *numstr* is the string-number conversion predicate, $=_s$ and $=_n$ denote equality over strings and natural numbers respectively, and $<_n$ is the natural number comparison (less-than) predicate.

With this definition and the results of Theorem 3 I can now show undecidability of the theory $T_{L,n,c}$ in Theorem 4.

Theorem 4. *The satisfiability problem for the theory $T_{L,n,c}$ is undecidable.*

Proof. The proof is shown via a recursive reduction from the theory T_p (Büchi's power arithmetic) to theory $T_{L,n,c}$, i.e., any quantifier-free formula in T_p can be equisatisfiably reduced to a quantifier-free formula in $T_{L,n,c}$. Thus, if the satisfiability problem for $T_{L,n,c}$ is decidable then so is the satisfiability problem for T_p. By Büchi's theorem [20] the satisfiability problem for T_p is undecidable, and hence so is the satisfiability problem for $T_{L,n,c}$.

The Reduction from T_p to $T_{L,n,c}$. We reduce each constant, function, predicate, and atomic formula of T_p to $T_{L,n,c}$ by applying the following rules recursively over the input formula:

1. Each natural number in $\mathbb{N}$ is represented directly as a constant in $T_{L,n,c}$.
2. Variables in T_p are represented directly as variables of numeric sort in $T_{L,n,c}$.
3. Addition of two terms $t_1 + t_2$ is represented directly as addition over natural numbers, $t_1 + t_2$, in $T_{L,n,c}$.
4. Equality of terms in T_p is represented directly via a recursive reduction as equality $t_1 =_n t_2$ of terms of numeric sort.

5. The less-than predicate in T_p is represented directly as comparison of natural numbers, $t_1 <_n t_2$.

6. The predicate $\pi(p, x, y)$ is expressible as follows: $\exists z : str, \exists x_s : str : (\text{“0”} \cdot z = z \cdot \text{“0”} \wedge len(z) = y \wedge numstr(p, x_s \cdot z) \wedge numstr(x, x_s))$. The interpretation of the π predicate is $p = x \times 2^y$. The variables z and x_s are string variables, and z is a string of the “0” character of length equal to y. The x_s variable is the string binary representation of the natural number x. The concatenation of x_s followed by z is a binary representation of p. It is easy to verify that the given formula over free numeric variables x, y, p is satisfiable iff $\pi(p, x, y)$ is satisfiable.

The reduction can easily be extended to arbitrary quantifier-free formulas in T_p. It is easy to verify that the reduction is sound, complete, and terminating for all inputs. □

This result is quite surprising, as *numstr* is not generally thought of as a powerful operator – certainly not as powerful as, say, *replaceAll*. However, the fact that string-number conversion can be used to encode arbitrary multiplication in this theory, as I just demonstrated, highlights the deep and subtle power of string-number conversion and suggests a potential source of complexity for automated reasoning about this operator.

3.1.3 Expressibility of $\pi(x, y, z)$

In this section I establish that the $\pi(p, x, y)$ and *numstr* predicates are expressible in terms of each other. I define a new theory T_π (different from T_p), which is the same as $T_{L,n,c}$ except that *numstr* is removed and replaced by the $\pi(p, x, y)$ predicate. From Section 3.1.2 it is clear that any formula involving the $\pi(p, x, y)$ predicate can be reduced to some formula in $T_{L,n,c}$ using some Boolean combination of *numstr* predicate, string equations, and length function. This shows that a reduction exists from T_π to $T_{L,n,c}$. I now show that a reduction in the opposite direction exists; that is, the *numstr* predicate can be expressed in terms of quantified formulas over the $\pi(p, x, y)$ predicate, word equations, and length function.

The value of these two recursive reductions is that it suggests that the π predicate is expressible using string equations and length function iff *numstr* is. Expressibility results are very useful tools in constructing reductions, distinguishing the expressive powers of various theories, and establishing (un)-decidability results. Additionally, these expressibility results suggest that the *numstr* predicate is much more complex, both from a theoretical and a practical point of view, than it seems at first glance.

Definition 5. *A predicate P is* ***expressible*** *in some theory T having language L_T if there exists an L_T-formula $\phi(x_1, \ldots, x_n)$ such that for all interpretations $m_1, \ldots, m_n$ of $x_1, \ldots, x_n$ allowed by T and such that $\phi(m_1, \ldots, m_n)$ is well-sorted, it follows that $P(m_1, \ldots, m_n)$ is true iff $\phi(m_1, \ldots, m_n)$ is true.*

The fact that $\pi(p, x, y)$ is expressible in terms of $numstr(i, s)$ in the theory $T_{L,n,c}$ follows immediately from the reduction from T_p to $T_{L,n,c}$ used to establish the undecidability theorem in the previous section. It remains to show the reverse direction, i.e., that $numstr(i, s)$ is expressible in terms of $\pi(p, x, y)$. [2]

Theorem 6. *$numstr(i, s)$ is expressible in terms of $\pi(p, x, y)$ in T_π.*

Proof. We represent $numstr(i, s)$ as a formula that asserts the non-existence of a witness for one of two kinds of error in the conversion. The first kind of error relates to the maximum possible value of i. Suppose s is a binary string of length n. Then s cannot represent a natural number greater than or equal to 2^n. The second error is a discrepancy between the binary representation of i and the binary string s. To check bit t of the number i, decompose i into $h2^{t+1} + x2^t + l$ where x is the tth bit of i and so $x = 0 \vee x = 1$, and l is the numeric representation of bits $t - 1$ through 0 and so $l < 2^t$. Then if $x = 0$ and $s[len(s) - 1 - t] =$ "1", or if $x = 1$ and $s[len(s) - 1 - t] =$ "0", there is an error. This gives us the following sentence:

$$
\begin{aligned}
numstr(i, s) \iff\ & \forall n, p, t, h, p_h, x, p_x, l, l_u, s_h, s_x, s_l : \\
& \neg(len(s) = n \wedge \pi(p, 1, n) \wedge i \geq p) \\
& \wedge \neg(\pi(p_h, h, t + 1) \wedge \pi(p_x, x, t) \\
& \wedge i = p_h + p_x + l \wedge \pi(l_u, 1, t) \wedge l < l_u \\
& \wedge s = s_h \cdot s_x \cdot s_l \wedge len(s_l) = t \wedge len(s_x) = 1 \\
& \wedge ((x = 0 \wedge s_x = \text{“1”}) \vee (x = 1 \wedge s_x = \text{“0”})))
\end{aligned}
$$

We can apply this rule recursively to the input formula, along with similar rules to the ones presented previously, to obtain a reduction from $T_{L,n,c}$ to T_π. □

[2] Note that I do not present a reduction from $T_{L,n,c}$ to T_p. However, I conjecture that one exists, due to the possibility of mapping the countably infinite set of string constants onto the countably infinite set of natural numbers and then constructing string functions and predicates as operators over natural numbers.

3.2 Decidability of the Theory of Strings with Length and Regular Expression Constraints

In this section, I present an elementary automata-theoretic decision procedure for the quantifier-free first-order theory T_{LRE} of strings, linear integer arithmetic, string length, and regular expression membership predicate.

The structure of the theory T_{LRE} is given as $\langle \Sigma^*, \mathbb{N}, RE, +, len, \in, =_n, <_n \rangle$ where Σ^* is the set of all string constants, $\mathbb{N}$ is the set of all natural numbers, RE is the set of all grounded regular expressions, $+$ is the two-operand addition function for natural numbers, len is the string length function, $\in$ is the regular expression membership predicate, and $=_n$ and $<_n$ are the natural number equality and comparison predicates. Note that this theory does not include concatenation of strings, nor does it include equality between strings (word equations).

I demonstrate that the satisfiability problem for T_{LRE} is decidable by illustrating a decision procedure. Although the decidability of this theory is implied by other results [45, 6], the construction of a decision procedure I show here is novel and is based on elementary principles of automata theory. This decision procedure is also a key inspiration for the practical regular expression solver I describe in Chapter 5. The proof in this section outlines a simple, constructive, and intuitive automata-based approach to simultaneously reason about regex and linear integer arithmetic constraints.

Theorem 7. *The satisfiability problem for the quantifier-free first-order theory T_{LRE} is decidable.*

Proof. Decidability is shown by describing a decision procedure for T_{LRE} and demonstrating that it is sound, complete, and terminating. The input to the decision procedure is a conjunction of (possibly negated) atoms, each of which is either:

- a regular language constraint, $S \in R$, for a string variable S and constant regular expression R
- a linear arithmetic constraint of the form $c_1 v_1 \pm c_2 v_2 \pm \ldots \pm c_k v_k \pm c_{k+1} len(S_1) \pm c_{k+2} len(S_2) \pm \ldots \pm c_{k+n} len(S_n) \bowtie C$, where each v_i is an integer variable, each term $len(S_i)$ represents the length of a string variable, C and each c_i are integer constants, and $\bowtie$ stands for either equality, disequality, or inequality.

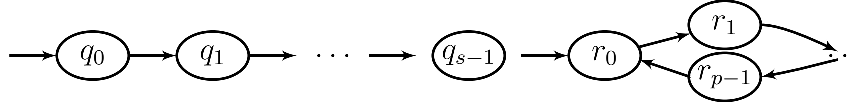

Figure 3.1: General form of length automaton with stem length s and period length p. All transitions are on the symbol '1'.

Consider each string variable S_i which appears in one or more regular language constraints. The decision procedure first constructs the product automaton $\mathbb{R}_i$ corresponding to the intersection of all regular languages over which S_i is constrained, that is, $\mathbb{R}_i = \bigcap_{x=1}^{n} R_x \cap \bigcap_{y=n+1}^{n+m} \overline{R_y}$ if $S \in R_1$, $S \in R_2$, etc. and $S \notin R_{n+1}$, $S \notin R_{n+2}$, etc. Note that any value of S_i which is in the language $\mathbb{R}_i$ automatically satisfies all regular language constraints on S_i, because $\mathbb{R}_i$ is the intersection of all such languages. Conversely, if $\mathbb{R}_i$ is empty, the decision procedure can immediately return UNSAT, since this implies that there is no string which simultaneously satisfies all regular language constraints on S_i.

After constructing each product automaton $\mathbb{R}_i$, the decision procedure constructs the corresponding **unary automaton** $\mathbb{L}_i$ by replacing all letters in the alphabet of $\mathbb{R}_i$ with the character '1' and otherwise leaving the rest of the automaton unchanged. Now, $\mathbb{L}_i$ is an automaton whose language is the set of all unary numbers corresponding to the lengths of all strings in $\mathbb{R}_i$. In general, this automaton is non-deterministic and may have ϵ-moves. The decision procedure then constructs a deterministic finite automaton $\mathbb{L}'_i$ from $\mathbb{L}_i$ and minimizes it. From a result shown by Eilenberg [29], $\mathbb{L}'_i$ must have the form illustrated in Figure 3.1 with some states $q_0, \ldots, q_{s-1}, r_0, \ldots, r_{p-1}$ final and where the values s and p are the "stem" and "period" of the automaton. The same result from [29] also gives that the unary numbers accepted by this automaton correspond to a finite union of arithmetic progressions. These progressions may be found as follows: for each final state $q_i \in \{q_0, q_1, \ldots, q_{s-1}\}$, the corresponding unary number 1^i is accepted by the automaton, which in turn corresponds to the integer i (which is trivially an arithmetic progression with period 0); and for each final state $r_j \in \{r_0, r_1, \ldots, r_{p-1}\}$, the unary numbers of the form $1^s(1^p)^*1^j$ are accepted by the automaton, which correspond to the integers enumerated by the arithmetic progression $(s + j) + np$ with $n \geq 0$.

From each automaton $\mathbb{L}'_i$ the decision procedure derives a finite collection of arithmetic progressions, the solutions of which in union are all possible valid lengths of a string $S_i \in \mathbb{R}_i$. Furthermore, as each arithmetic progression is of the form $a + nb$, for constant a, b and a fresh integer variable n, the implied length constraints can be expressed as a collection of terms in Presburger arithmetic. From here, the decision procedure can

take the original system of length constraints and augment it with the implied length constraints for each string variable as follows. For each string variable S_i, suppose the set of arithmetic progressions is given by $L_i = \{a_1 + n_1 b_1, a_2 + n_2 b_2, \ldots, a_x + n_x b_x\}$. Then add the disjunction of atomic formulas given by $(a_1+n_1b_1)-len(S_i) = 0 \vee (a_2+n_2b_2)-len(S_i) = 0 \vee \ldots \vee (a_x + n_x b_x) - len(S_i) = 0$. Additionally, add a constraint of the form $n_i \geq 0$ for each fresh variable n in each arithmetic progression; and add a constraint of the form $len(S_i) \geq 0$, since the length of any string is non-negative by definition. The resulting augmented system of integer constraints (including string length terms) is a system of quantifier-free Presburger inequalities, so at this point the decision procedure can solve it and either find that there are no solutions, or that there is some solution. If the augmented system has no solutions, this implies that there are no lengths of strings which satisfy the original length constraints plus the length constraints implied by the regular language membership constraints, so the decision procedure returns UNSAT. Otherwise, it finds a solution which gives (in particular) a value for each length term $len(S_i)$. Because the solution to the augmented system satisfies one of the terms in each disjunction of arithmetic progressions, the value of $len(S_i)$ in the solution corresponds to the length of a solution in $\mathbb{R}_i$. Now for each S_i, it can find *any* string having this length in $\mathbb{R}_i$, e.g. by depth-first search over the automaton, for each S_i. This solution satisfies all regular language constraints on S_i because of how $\mathbb{R}_i$ is constructed, and additionally satisfies all length constraints on $len(S_i)$. At this point the decision procedure returns SAT and has produced a certificate consisting of the solution to the augmented system of length constraints plus the string value chosen for each S_i.

It is easy to see that the above algorithm always terminates, given a decision procedure for quantifier-free Presburger arithmetic. I demonstrate soundness and completeness of the above algorithm as follows. There are two places in which this algorithm can return UNSAT: either upon finding that some intersection of regular expression constraints $\mathbb{R}_i$ is empty, or upon finding that the augmented system of length constraints has no solutions. By the construction of $\mathbb{R}_i$ as an intersection of all regular language constraints on each corresponding string variable, if this intersection is empty then the original constraints cannot all be satisfied simultaneously, hence the input formula is unsatisfiable. If the augmented system of length constraints has no solutions, then by the construction of the length automaton $\mathbb{L}'_i$ there is no string whose length satisfies the length constraints given in the input formula while simultaneously satisfying the implied length constraints from the regular expression terms, hence the input formula is unsatisfiable in this case as well. Similarly, as already illustrated in the description of the algorithm, when the algorithm returns SAT, the solution to the system of length constraints plus the string values S_i derived from these length values and the intersection automata $\mathbb{R}_i$ satisfy the original

formula, hence the input formula is satisfiable. Therefore, the algorithm presented above is sound, complete, and terminating, hence it is a decision procedure for the quantifier-free theory of regular language constraints and Presburger arithmetic over the lengths of words.

The decision procedure is also in PSPACE, as the construction of all necessary automata, computation of their intersections, and construction of length constraints can be performed in PSPACE, solving the system of arithmetic constraints can be performed in PSPACE (in fact this is NP-complete), and production of the satisfying string assignments once the arithmetic constraints have been solved can also be done in PSPACE. Therefore, the decision problem for the quantifier-free theory of regular language constraints and arithmetic on length constraints is PSPACE-complete, because the equivalence problem for finite automata trivially reduces to the decision problem for this theory, for which I have just shown a PSPACE decision procedure. □

Chapter 4

Z3str4: A Solver for Theories Over Strings

In this chapter, I describe the architecture and implementation of the Z3str4 string solver. I begin in Section 4.1 by motivating the algorithm selection approach and explaining why diverse solving algorithms are key to achieving greater performance in solving string constraints. I then illustrate the architecture of Z3str4 in Section 4.2, highlighting the different algorithm combinations that it uses and showing the control flow and data flow between different modules in the implementation. I explain the procedure by which Z3str4 performs algorithm selection in Section 4.3. I also explain several of the novel algorithms in Z3str4, including the length abstraction solver (LAS) in Section 4.4, the fixed-length model construction procedure in Section 4.5, theory-aware heuristics related to string solving in Section 4.6, and the clause sharing heuristic in Section 4.7. I conclude the chapter by presenting a detailed empirical evaluation of Z3str4 against other state-of-the-art string solvers in Section 4.8.

4.1 Motivation

As is evident from both theory and practice, reasoning about the quantifier-free first-order theory of word equations, functions such as `concat` and `substring`, predicates such as `contains`, regular expression membership constraints, string-integer conversion functions, and linear integer arithmetic over string length is hard (see Chapter 3). Despite these difficulties, much research has been done on practical algorithms for solving string

constraints obtained from many real-world analysis, testing, verification, and synthesis applications [59, 30, 68, 46]. Examples of such solvers include HAMPI [41], Stranger [71], Z3's sequence solver (Z3seq) [27], CVC4 [44], Norn [6], Trau [4], S3 [67], and the Z3-str family of solvers upon which Z3str4 is based, whose predecessors include Z3str2 [72] and Z3str3 [15]. Each tool has varying strengths and weaknesses. Precisely because solving string formulas is hard in general – and is still hard or even undecidable for many specific theory combinations relevant to program analysis – solver designers have come up with a diverse set of practical algorithms that incorporate a variety of tradeoffs. Some of these methods work well for pure word equations, but not so well for integer constraints over string length. Other methods work well for a mix of word equations and integer constraints, but perform poorly on more complicated constraints involving functions such as `substring` or predicates like `contains`. This diversity of algorithms for solving string constraints immediately presents an opportunity from an *algorithm selection* perspective. The key challenges involved in algorithm selection for string instances relate to predicting the performance of a given algorithm on a particular input formula. So-called "features" of formulas which correlate well with solving difficulty or algorithm performance have been studied for some time in the case of Boolean satisfiability solving (SAT), and algorithm selection techniques have shown strong results when trained against empirical hardness models for SAT solvers [69]. Algorithm selection techniques have also been applied using machine learning algorithms over different standalone SMT solvers [56]. As feature identification and hardness models have not been developed specifically for theories over strings, development of Z3str4 has provided an opportunity to approach a better understanding of what might make some string instances harder to solve than others, and how different algorithms could be developed to handle different kinds of inputs.

4.2 Architecture of Z3str4

In this section, I describe the architecture of Z3str4 and show how its components and novel algorithms work together at a high level. An architectural diagram of Z3str4 is shown in Figure 4.1, illustrating its control and data flow.

Input to the Z3str4 solver is given as a formula in SMT-LIB syntax as described in Chapter 2, and the output is one of SAT, UNSAT, or UNKNOWN. Z3str4 is built on top of Z3 and reuses its parser and core architecture. Once parsed, the formula is then passed on to Z3str4's arm selection procedure, which is described in detail in Section 4.3. This procedure makes use of a series of static "probes" to analyze the formula and decide which of its arms is the most appropriate for the given input. Each arm calls the length

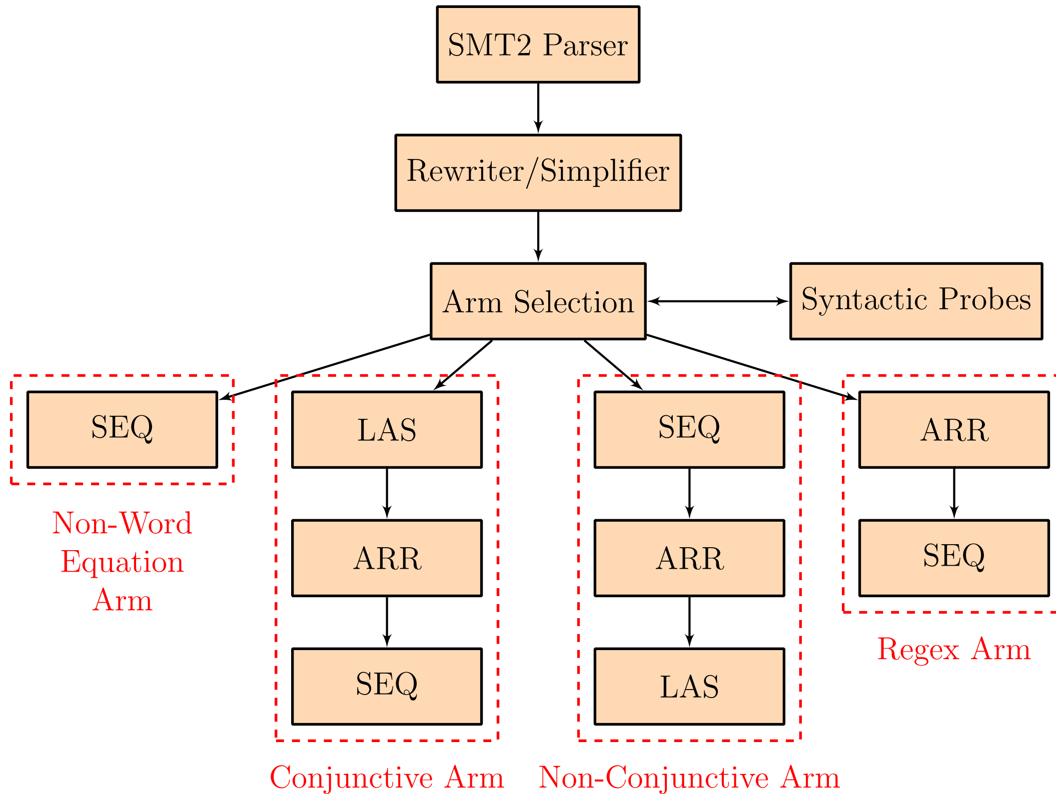

Figure 4.1: Architecture of the Z3str4 tool.

abstraction solver (Section 4.4), the arrangement solver (Section 4.5), and Z3's sequence solver in some predetermined order, as shown in Figure 4.1. The method by which the probes perform arm selection is described in more detail in Section 4.3.

The length abstraction solver and arrangement solver are able to share certain learned constraints between invocations, even though they are called independently. This allows solvers to benefit from the work done by earlier ones in the sequence determined by an arm. I discuss this mechanism in Section 4.7.

4.3 Algorithm Selection

In this section, I describe the arm selection procedure used by Z3str4 to choose the series of algorithms that are applied to solve an input formula.

Algorithm 1: Z3str4's arm selection procedure.

```
   Data: formula φ of quantifier-free string and arithmetic constraints in conjunctive
         normal form
 1 φ ← Simplify(φ)
 2 if φ contains regex constraints then
 3 |   ArrangementSolver()
 4 |   SequenceSolver()
 5 else
 6 |   if majority of top-level formulas in φ are word equations then
 7 |   |   if ConjunctiveFragment(φ, TRUE) then
 8 |   |   |   LengthAbstractionSolver()
 9 |   |   |   ArrangementSolver()
10 |   |   |   SequenceSolver()
11 |   |   else
12 |   |   |   SequenceSolver()
13 |   |   |   ArrangementSolver()
14 |   |   |   LengthAbstractionSolver()
15 |   |   end
16 |   else
17 |   |   SequenceSolver()
18 |   end
19 end
```

The arm selection method uses static features of the instance to determine which of the three solver algorithms – the length abstraction solver, the arrangement solver, and Z3's sequence solver – are invoked and in what order. The pseudocode for the arm selection procedure is shown in Algorithm 1, and the conjunctive probe subroutine is illustrated in Algorithm 2. The conjunctive probe subroutine was developed by Federico Mora [50].

The input formula ϕ is first passed to Z3's simplifier and term rewriting procedure (line 1). The algorithm selection procedure then follows a three-tiered sequence of checks for static features (lines 2–19). This sequence of checks is illustrated in Figure 4.2. The order and choice of solvers to use was determined by a combination of empirical results and experimentation. For example, by observing that the arrangement solver has superior performance on industrial instances with many regex constraints, I determined that it would be preferable for Z3str4 to leverage it when regex constraints are present in general. The sequence of decisions made is as follows. First, if any regex constraints appear in the input formula ϕ, the arrangement solver is used. Otherwise, if a majority of top-level formulas

Algorithm 2: Z3str4's conjunctive fragment probe.

```
1  Subroutine ConjunctiveFragment
2  |
   Input  : formula φ to check, Boolean sign indicating polarity of φ
   Output : TRUE if φ is in the conjunctive fragment and FALSE otherwise
3  while φ matches Not(ψ) do
4  |   sign ← ¬(sign)
5  |   φ ← ψ
6  end
7  if φ matches X = Y and sign is FALSE and X is a string term then
8  |   return FALSE
9  end
10 if (φ matches prefixof(X, Y) or φ matches suffixof(X, Y)) and sign is FALSE then
11 |   return FALSE
12 end
13 if φ matches contains(X, Y) or φ matches regexIn(X, R) or φ matches
     replace(X, Y, Z) then
14 |   return FALSE
15 end
16 for a ∈ arguments of φ do
17 |   if ConjunctiveFragment(a, sign) = FALSE then
18 |   |   return FALSE
19 |   end
20 end
21 return TRUE
```

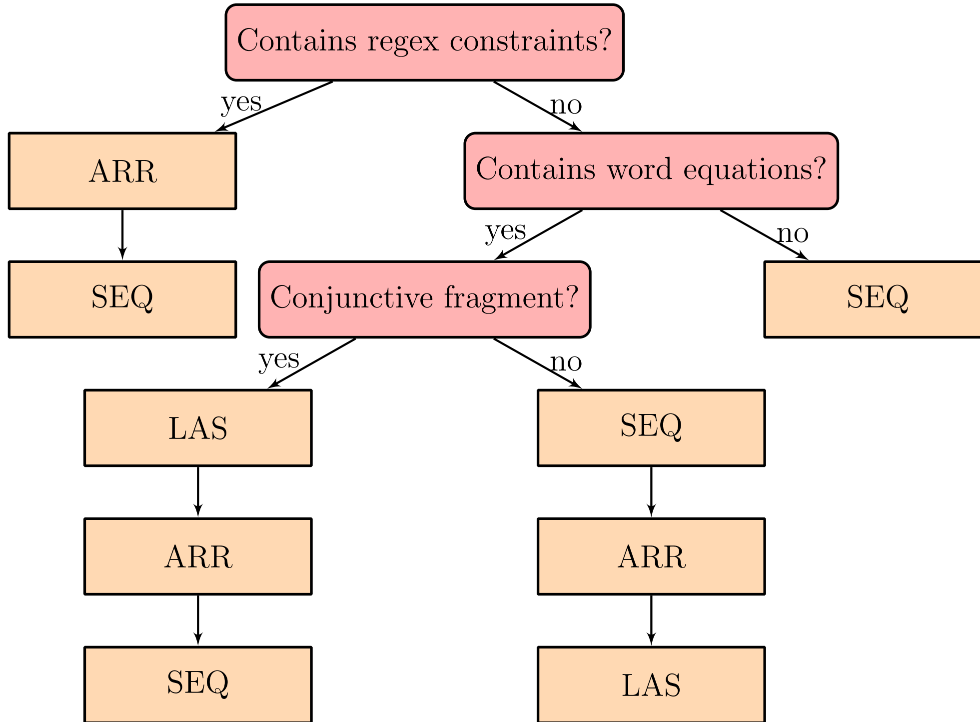

Figure 4.2: The three-tiered probe used by Z3str4 to perform arm selection.

in the input are *not* word equations, the sequence solver is used. Finally, the algorithm selection procedure calls the `ConjunctiveFragment` subroutine on line 7. This subroutine traverses the AST of ϕ and checks whether any of the following expressions appear in the input formula: string disequalities, negated `str.prefixof` and `str.suffixof` predicates, or any `str.contains`, `str.replace`, or regex membership terms. The intuition used here is that the terms checked for are relatively hard for the bit-vector solver to handle after `ReduceToBV` is applied, due to disjunctions of bit-vector constraints in the reduction. (I explain this in more detail in Section 4.5. For example, `str.contains` reduces to a disjunction of equalities between characters under a fixed-length interpretation.) Thus, if the probe returns TRUE, Z3str4 can call the length abstraction solver first with the expectation that it will exhibit the best performance, followed by the arrangement solver and the sequence solver. Otherwise, it will call the sequence solver first, followed by the arrangement solver and LAS.

The procedure uses the information from the probe to choose between multiple different arms that are used to solve the formula. An "arm" here refers to a fixed ordering of the

following algorithms: the length abstraction solver, discussed in Section 4.4; the arrangement solver, discussed in Section 4.5; and Z3's sequence solver. The arm configuration specifies how these algorithms are called and in which order. Each solver is called with a static set of parameters and a timeout that is computed based on the total time limit given to the tool. The first solver in the selected arm that successfully answers SAT or UNSAT causes the procedure to return that answer and terminate. If a solver times out or gives up, the next solver in the arm is called. If no solver returns an answer, the procedure returns UNKNOWN if there was time remaining or TIMEOUT otherwise.

4.4 Length Abstraction Solver

In this section, I briefly describe the length abstraction solver used by Z3str4. [1]

The length abstraction solver (LAS) takes in a conjunction ϕ of string literals, and either returns an assignment that satisfies ϕ, returns UNSAT, or returns UNKNOWN. As the algorithm operates on an approximation of length constraints, it is prohibitively expensive to allow too many "guesses" to be made. A maximum iteration count is fixed when the algorithm begins, and if this count is reached before either finding a satisfying assignment or finding UNSAT, the algorithm terminates and returns UNKNOWN. LAS begins by calling `MultisetCheck` at line 1. This subroutine can quickly determine UNSAT for many kinds of string constraints, described in more detail in Section 4.4.1. If this check does not determine that the input is UNSAT, then LAS constructs $\hat{L}$, a length abstraction over the input string constraints in ϕ, and enters the main solving loop. Each iteration of the loop checks the satisfiability of the current abstraction $\hat{L}$. If this is ever unsatisfiable, then the implicit and explicit integer constraints in ϕ are also unsatisfiable, and the algorithm returns UNSAT. Otherwise, a candidate model M_L is found for $\hat{L}$ and this model is used to construct a system of fixed-length bit-vector character constraints for the string constraints in ϕ by the methods described in Section 4.5. This system is denoted as ϕ_{bv}. The algorithm then checks the satisfiability of ϕ_{bv}. If it is satisfiable, the satisfying assignment to this system can be translated back into a model for the string constraints in ϕ, and the algorithm returns SAT. Otherwise, the algorithm increments an iteration counter, terminating if this exceeds a maximum number of iterations, and otherwise learning a conflict clause ψ blocking the current length assignment in $\hat{L}$ and looping. The limit on iterations is necessary to ensure that the algorithm always terminates. As a result, LAS is inherently incomplete, but this is not a drawback in practice – Z3str4 never calls

[1]The LAS algorithm was developed by Federico Mora [50], and I illustrate it here for the sake of a complete presentation of Z3str4.

Algorithm 3: Z3str4's length abstraction solver (LAS).

Data: Conjunction of theory literals ϕ
Result: SAT, UNSAT, or UNKNOWN

```
1  if MultisetCheck(φ) returns FALSE then
2  |   return UNSAT
3  end
4  L̂ ← length abstraction over φ
5  while L̂ satisfiable do
6  |   M_L ← candidate arithmetic model for L̂
7  |   φ_bv ← fixed-length bit-vector constraints for φ with respect to M_L
8  |   if φ_bv satisfiable then
9  |   |   M_S ← string model from satisfying assignment to φ_bv
10 |   |   return SAT
11 |   else
12 |   |   iteration count += 1
13 |   |   if maximum iteration count reached then
14 |   |   |   return UNKNOWN
15 |   |   end
16 |   |   ψ ← learned conflict clause from φ_bv
17 |   |   L̂ ← L̂ ∧ ψ
18 |   end
19 end
20 return UNSAT
```

LAS by itself, and it is always followed by either the arrangement solver or the sequence solver. The purpose of LAS is to be an efficient "pre-solver" for instances where the fixed-length reduction is expected to perform well without much overhead (in other words, for instances in the conjunctive fragment, as described above). LAS operates at its best for instances that can be solved without needing the full power of the sequence solver or arrangement reduction, and for instances where the sequence solver or arrangement solver might encounter too much overhead in reasoning about many subformulas (for example, if the arrangement solver needs to process a large number of arrangements in order to make any progress).

4.4.1 MultisetCheck Subroutine

The subroutine `MultisetCheck` is a heuristic that analyzes a simple static property of atomic string formulas, and returns false if these formulas are unsatisfiable based on this. The check performed here constructs the Parikh image [53] of a word equation with respect to the characters and variables that it contains and checks whether it is consistent. As an illustrative example, consider the word equation $a \cdot X = X \cdot b$. In order for these strings to be equal, the number of occurrences of each character on the left-hand side must be the same as the number of occurrences on the right-hand side. Therefore, if some character has a different number of occurrences on either side of an equality, then the given terms cannot be equal to each other. Consider the character "a". On the left-hand side of this equation, the number of occurrences of "a" is one, for the constant string, plus however many times it occurs in X – call this X_a. On the right-hand side, the number of occurrences of "a" is just X_a. If some model for X satisfied $a \cdot X = X \cdot b$, it must also satisfy the abstraction $1 + X_a = X_a$ for however many occurrences of "a" it contained. However, it is easy to check that this abstraction has no solution for X_a. There is no way to assign X such that both sides have the same number of "a"s, and therefore the original word equation is unsatisfiable.

4.5 Fixed-Length Model Construction

In this section, I describe extensions that I made to the arrangement solver in order to integrate a powerful and efficient bit-vector backend for solving character constraints and constructing models for string formulas. Z3str4 uses this reduction as part of an abstraction-refinement loop to handle low-level string formulas and integrates with the arithmetic

solver to refute candidate length assignments that have no valid solutions over the string constraints.

During the process of reasoning in an SMT solver, if an input formula is believed to be satisfiable, each theory solver must construct a model for it. With respect to strings, a model is a mapping from string variables to string constants such that all string constraints (word equations and string predicates) are satisfied with respect to the Boolean abstraction explored by the core solver. Additionally, the model must satisfy constraints in other domains, in particular arithmetic constraints that involve the lengths of strings. Although arithmetic constraints are not handled directly by the string theory solver, it must nevertheless be aware of them and avoid constructing models that contradict known facts about the lengths of strings.

Previous versions of Z3str3, as well as Z3-str and Z3str2, used a naïve brute-force method to construct models over string variables during the final phase of the search. The method performed a linear search over all possible lengths n of each string variable, starting from 0, and a second linear search over all possible string constants of length n for each string variable, starting from "a^n" and trying all possibilities in increasing lexicographical order. This algorithm is extremely inefficient for several reasons. First, the linear search for lengths of string variables proceeds independently of facts that might be known by the arithmetic solver, such as potential lower/upper bounds on the lengths of strings. This can mean that the string solver tries length assignments that have already been shown to be unsatisfiable. Second, the search over string constants of a fixed length has exponential complexity, as there are 256^n possible ASCII strings of length n (and this is even worse for other alphabets, such as Unicode). Solving even trivial formulas whose solutions have strings of lengths longer than a few characters – for example, $len(X) > 100$ – would time out. Third, each string constant is searched for independently, regardless of any facts that may be known about the variables that are involved. For example, if A is known to be a prefix of B, the method used here will still consider all possible models for A and B, including models where the string constant assigned to A is not a prefix of the one assigned to B. Finally, since individual models are checked one at a time, if a model is found to be in conflict with the asserted constraints, the string solver is extremely limited in what it can learn from this conflict in order to prune the search space. Only one possible model can be blocked at a time, when it might be more desirable to block larger parts of the search space if more can be learned from a given conflict.

In identifying the limitations of this existing approach, it is clear that a more sophisticated model construction algorithm is necessary, and that it should take advantage of both length constraints and string constraints as well as leveraging facts about conflicts to construct richer conflict clauses to prune the search space. This brings to mind tech-

niques such as the fixed-length unfolding approach used by foundational string solvers such as HAMPI [41]. As previously mentioned, approaches to string constraint solving can be classified into automata-based, word-based, and unfolding-based techniques [8]. Tools such as HAMPI which use unfolding-based approaches often have a limitation wherein the maximum length of each string variable must be fixed *a priori*. However, unfolding-based approaches allow reasoning directly over the characters that make up each string variable, which is very efficient. The model construction architecture I describe and implement in this chapter is a hybrid approach that combines the efficiency of an unfolding-based strategy (reduction of fixed-length string equations to bit-vectors) with the generality of a word-based algorithm to reason about unbounded strings. This idea is similar to principles explored by Karhumäki et al. [39], in particular the concept of "unfixed parts" of string variables which can be filled arbitrarily when constructing a solution.

4.5.1 Solving Strings via Arrangements

I first briefly describe the "arrangement" algorithm for solving string constraints. This algorithm was first implemented in the original Z3-str solver [74, 72] by Zheng, Zhang, and Ganesh, upon which Z3str4 (and Z3str3) are based.

The core idea behind the arrangement technique is the reduction of string equations to simpler string equations until the formula is in a so-called "solved form", wherein every string variable appears on one side of an equality by itself and the other side is either a string constant, a single variable, or a concatenation of two variables. When considering an equality between strings, the arrangement technique introduces a disjunction of formulas describing the possible relationships between variables on the left-hand side and right-hand side of the equation. For example, consider the equality $A \cdot B = X \cdot Y$. In this formula, there are three possible relationships between A and X: either A and X have the same length, or A is shorter than X, or A is longer than X [2]. These possible arrangements are illustrated in Figure 4.3. The arrangement technique expresses the possible relationships with the following implied formulas. In the first case, $A = X$ and $B = Y$. In the second case, $A \cdot X_1 = X$ and $B = X_1 \cdot Y$, for a fresh string variable X_1. In the third case, $A = X \cdot X_2$ and $X_2 \cdot B = Y$, for a fresh string variable X_2. These three formulas are asserted as a disjunction to the core solver, which chooses one branch to explore. The solver reduces the resulting formulas even further until no more reductions are possible, at which point the formula is in solved form.

[2]This is a restatement of a fact also referred to as Levi's Lemma [43].

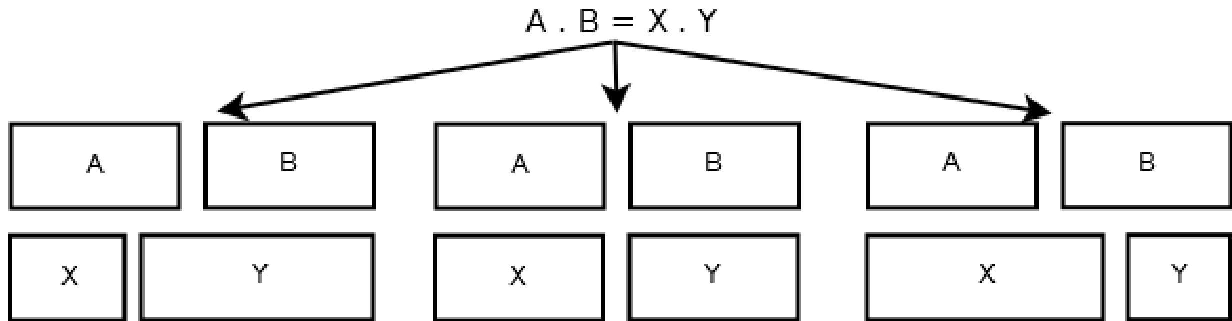

Figure 4.3: Possible arrangements of $A \cdot B = X \cdot Y$.

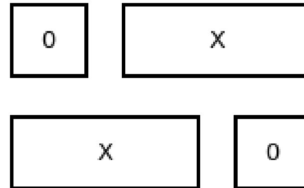

Figure 4.4: Illustration of an overlapping variable X in $0 \cdot X = X \cdot 0$.

An important weakness of Z3str3's arrangement solver is that it cannot handle word equations which have the same variable occurring on both the left-hand and right-hand side of an equation, referred to as an "overlapping variable". Consider the equation $0 \cdot X = X \cdot 0$, as illustrated in Figure 4.4. Z3str3's arrangement solver would detect the existence of an overlapping variable and return UNKNOWN because X appears on both sides. The reason for not handling these equations is as follows. Suppose that $len(X) > 1$; then the arrangement constructed is $0 \cdot X_1 = X$ and $X = X_1 \cdot 0$, for a fresh variable X_1, following the pattern of the second case described above. Since these terms are both equal to X, we now have the word equation $0 \cdot X_1 = X_1 \cdot 0$. However, observe that this is identical to the equation we started with except that X has been replaced with X_1. If the solver continues generating such arrangements, it can enter an infinite loop and become unable to make any further progress. Thus, the original design of Z3str2 incorporated an algorithm to detect such arrangements which contained overlapping variables and terminate the search upon encountering them; Z3str3 includes the same algorithm. Handling such equations is an additional motivation for using the more sophisticated reduction to bit-vector character constraints. Observe that once string variable lengths have been fixed, any arrangement with overlapping variables is now merely a bit-vector equation and an appropriate bit-vector solver can be invoked on it to decide its satisfiability. For example, again considering $0 \cdot X = X \cdot 0$, if the arithmetic solver proposes the candidate model

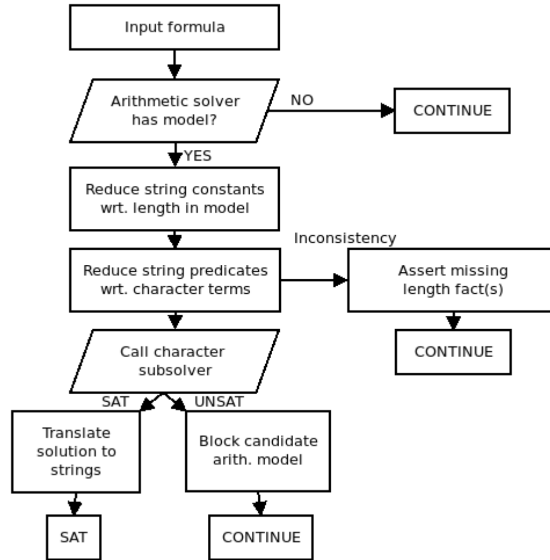

Figure 4.5: Architecture of Z3str4's fixed-length model construction algorithm.

$len(X) = 2$, the string-to-bit-vector reduction would reduce X to the 8-bit bit-vector characters x_1x_2 and solve the bit-vector equation $0x_1x_2 = x_1x_20$, finding it satisfiable with solution $X = 00$. Therefore, when implementing the bit-vector reduction described in the following sections, Z3str4 does not immediately return UNKNOWN when encountering overlapping variables, although it still detects them and stops constructing arrangements for such equations (as the problem of infinite reductions must still be handled). In other words, this hybrid approach allows Z3str4 to take advantage of a word-based reduction for as long as it is useful, and dynamically switch to an unfolding-based approach once no further progress can be made using the arrangement technique. The only compromise that must be made is that the formula may not be in solved form once the arrangement reduction completes; in practice, this is not a limitation for the bit-vector solver, for which this was never a requirement.

4.5.2 A Bit-Vector Backend for Solving String Constraints

With the limitations and motivations clearly identified, I now describe the fixed-length model construction procedure used by Z3str4 for solving string constraints. This procedure is used in Z3str4's arrangement solver and LAS (length abstraction solver) as part of the final check performed before answering that a formula is satisfiable and outputting a model

for the string variables. The intuition behind this procedure is as follows. In a CDCL(T) SMT solver architecture, such as Z3, each theory solver is responsible for verifying whether the Boolean abstraction of the input formula is consistent with the semantics of the theory it handles, and if so, constructing a model that satisfies the input formula with respect to this theory. For instance, the core solver may set a Boolean literal corresponding to the equality $len(X) = len(Y)$ to true; it only sees a Boolean literal and does not understand the semantics of that equality. This is left to the theory solvers to handle. In this case, the integer theory solver must construct a model for $len(X)$ and $len(Y)$ consistent with this equality (if possible). In turn, the string theory solver must also construct a model for X and Y consistent with this. The key idea here is that the string solver can leverage the model constructed by the integer solver to guide its own model construction. The search space of all possible string assignments becomes finite once the length of each string is fixed. Although trying every possible solution would be theoretically feasible, this can be handled much more efficiently by using existing methods for finite domain reasoning, such as Z3's bit-vector solver. In this way, the string solver can rely on the arithmetic solver to handle the length constraints as though they were an abstraction of the string formula and attempt to refine this abstraction with respect to the string constraints, either discovering a satisfying assignment for the string variables with respect to this length assignment or asserting a conflict clause that blocks the current length assignment and continuing the search.

The core idea behind the bit-vector backend, illustrated in Algorithm 4, is the following: for a given string formula ϕ over which the arrangement reduction technique described previously has run to completion, the method first queries Z3's integer solver to obtain a consistent length assignment to all string variables in ϕ. Once such an assignment has been obtained, each string variable can be interpreted as a fixed-length ordered sequence of, for example, 8-bit bit-vectors each representing an ASCII character[3]. This immediately suggests a very natural reduction from strings to bit-vectors (such as HAMPI [41] uses). Z3str4 implements exactly such a reduction from strings to bit-vectors and uses Z3's native bit-vector solver to solve the resulting constraints. If Z3's bit-vector solver returns SAT, it follows that the input string constraints are also satisfiable. However, if it returns UNSAT, then Z3str4 blocks the corresponding length assignment and asks the core solver to continue the search, in the manner of conflict-driven clause learning. This allows the core solver and arithmetic solver to prune the search space further with respect to the blocked assignment. This process repeats until either Z3str4 converges to the correct answer and returns SAT, the core solver finds a top-level conflict and returns UNSAT, or the solver runs out of time

[3]New versions of Z3str4 also support Unicode via exactly the same procedure described here; the only difference is that the characters have more than 8 bits.

Algorithm 4: Z3str4's string-to-bit-vector algorithm

Input : formula ϕ of quantifier-free string and arithmetic constraints in conjunctive normal form
Output : SAT or CONTINUE

```
1  if arithmetic solver has no candidate model then
2  |  return CONTINUE
3  end
4  ψ ← candidate model from arithmetic solver
5  φ_bv ← empty formula
6  for f ∈ φ do
7  |  if f matches X = Y and X, Y are string terms then
8  |  |  X_bv ← bit-vector representation of X wrt. ψ
9  |  |  Y_bv ← bit-vector representation of Y wrt. ψ
10 |  |  if X_bv and Y_bv have differing widths then
11 |  |  |  assert X = Y → len(X) = len(Y)
12 |  |  |  return CONTINUE
13 |  |  end
14 |  |  φ_bv ← φ_bv ∧ X_bv = Y_bv
15 |  end
16 end
17 if φ_bv satisfiable then
18 |  ψ_bv ← model from bit-vector solver
19 |  for v ∈ Vars(φ) do
20 |  |  assert v = ψ_bv(v)
21 |  end
22 |  return SAT
23 else
24 |  assert ¬ψ
25 |  return CONTINUE
26 end
```

or resources. A schematic view of Algorithm 4 is also shown in Figure 4.5.

Algorithm 4 is called during the `final_check` callback which is invoked by Z3's core solver once no further propagation or branching needs to be done. First, the algorithm queries the arithmetic theory solver for a candidate model, that is, a satisfying assignment to every arithmetic term in the instance. If the arithmetic solver has not yet found such an assignment, it instructs the core solver to continue the search. Otherwise, model construction proceeds using the candidate arithmetic model. The algorithm fixes the length of every

string variable that appears in the current assignment using the candidate model provided by the arithmetic solver and reduces every string term to a fixed-length sequence of 8-bit bit-vectors representing the ASCII characters of that term. The algorithm performs this reduction for string equalities and disequalities, string predicates such as `str.contains`, and regex membership constraints. For string equalities and disequalities, both sides of the equation are reduced to character terms, then corresponding characters are asserted to be equal to each other. String predicates are reduced according to their semantics under a fixed-length character-based interpretation (for example, `str.contains` reduces to a disjunction of character equality predicates, each of which asserts that the characters of the strings being compared are equal at a different fixed position). For regex membership constraints, the algorithm reduces the regex to a nondeterministic finite automaton (NFA) and enumerates each path through the automaton of the appropriate length ending in an accepting state.

The bit-vector constraints are not asserted into the main context of Z3, but are instead asserted into an isolated sub-context that is solved independently. This is done in order to solve the system of bit-vector constraints without polluting the main string context with extraneous axioms and conflict clauses that are not needed after the search completes. Once all reductions have been performed, the algorithm queries the bit-vector sub-solver for a satisfying assignment. If the sub-solver returns SAT, the model can be translated back to a model over strings by concatenating the character assignments and mapping them back to the original string terms. This model is then asserted as a candidate solution to the main solver. If the sub-solver returns UNSAT, a conflict clause is asserted in the main solver that blocks the length assignment found by the arithmetic solver from being searched again.

A conflict can also occur if the length information provided by the arithmetic solver is in conflict with the axioms of strings. For example, a regex membership constraint may have no solutions of the given length. Should this occur, the algorithm asserts a conflict clause blocking the length-string constraint pair in the main solver, and continues the search without calling the bit-vector solver. Such conflict clauses are desirable because they are very short and directly relate to the facts that are in conflict, rather than blocking an assignment which may include terms unrelated to the actual conflict.

4.6 Theory-Aware Heuristics

In this section, I describe the theory-aware branching and theory-aware case split heuristics implemented in Z3str4. These heuristics involve changes to Z3's core solver, which han-

dles the Boolean structure of the formula and performs propagation and branching. The intuition behind so-called "theory-aware" heuristics is that in a CDCL(T) architecture, the core solver, which handles the Boolean abstraction, normally does so unaware of the semantics of the terms it is handling. For example, it can set a literal corresponding to the equality $len(X) = len(Y)$ to true while being unaware of the semantics of this equality. As previously mentioned, the arithmetic theory solver and string theory solver would be responsible for checking the consistency of this equality with the semantics of their respective theories. Furthermore, the amount of information that theory solvers can provide to the core solver to guide the search is usually very limited, normally limited to asserting axioms or implied facts and asserting that the current partial assignment is inconsistent. Theory-aware heuristics expand the ability of theory solvers to provide information to the core solver to guide the search in new ways. The heuristics I present here both modify the way that Z3's core solver computes branching information, which is crucial to implementing a performant tool and which normally is done without any interaction with theory solvers at all.

4.6.1 Theory-Aware Branching

Consider the case where the solver learns the equality $X \cdot Y = A \cdot B$ for non-constant terms X, Y, A, B. Z3str4, as described in Section 4.5.1, handles this equality by considering a disjunction of three possible arrangements:

Arrangement 1: $X = A$ and $Y = B$

Arrangement 2: $X = A \cdot s_1$ and $s_1 \cdot Y = B$ for a fresh non-empty string variable s_1

Arrangement 3: $X \cdot s_2 = A$ and $Y = s_2 \cdot B$ for a fresh non-empty string variable s_2

Of the three possible arrangements, the first is the simplest to check because it does not introduce any new variables and only asserts equalities between existing terms. Therefore, it is intuitively desirable for Z3's core solver to prioritize checking this arrangement before the others. This directly motivates the implementation of the **theory-aware branching** heuristic, which allows theory solvers to give certain literals increased or decreased branching priority in the core solver during the search. The advantage gained by theory-aware branching is the ability to give the core solver information regarding the relative importance of each branch, allowing the theory solver to exert additional control over the search. Simpler branches are always prioritized over more complex ones.

Theory-aware branching is implemented as a modification of the branching heuristic in Z3. The default branching heuristic in Z3 is activity-based, similar to VSIDS [51]. The core solver will branch on the literal with the highest activity that has not yet been assigned. Activity is increased additively when a literal appears in a conflict clause, and decayed multiplicatively at regular intervals. There has been some work in taking domain-specific knowledge into account in the context of branching heuristics and custom decision strategies [31, 52, 28] although this has as of yet not been applied to specific theory activities in an SMT solver.

The theory-aware branching technique computes the activity of a literal A as the sum of two terms A_b and A_t, wherein the term A_b is the "base activity", which is the standard activity of the literal as computed and handled by Z3's core solver. The term A_t is the "theory-aware activity". The value of this term is provided for individual literals by theory solvers, and is taken to be 0 if no theory-aware branching information has been provided. This modification causes the core solver to branch on the literal with the highest overall activity A, taking into account both the standard activity value and the theory-aware activity. Therefore, assigning a (small) positive theory-aware activity to a literal will cause it to have higher activity than usual, making it more likely for the core solver to choose it to branch on. Conversely, assigning a (small) negative theory-aware activity will deter the core solver from choosing that literal. Theory-aware branching in Z3str3 modifies the activities of theory literals as follows:

1. Literals corresponding to arrangements that do not create new variables (as in Arrangement 1 above) are given a large (0.5) A_t. Other arrangements in the same case are given a small (0.1) A_t.
2. Arrangements that allow a variable to become equal to a constant string are given a small (0.2) A_t.

The values of A_t were chosen to be similar in scale to the initial activity values assigned to literals by the default branching heuristic. Although this technique is currently used by the string solver component, theory-aware branching is also useful in many other contexts where new search paths may have unequal importance, such as non-linear arithmetic.

4.6.2 Theory-Aware Case Split

During the search, a theory solver can create terms which encode a disjunction of Boolean literals that are pairwise mutually exclusive, i.e., exactly one of the literals must be assigned

true and the others must be assigned false. This is referred to as a *theory case split*. As an example, consider the word equation $X \cdot Y = c_1 \ldots c_n$, where X, Y are string variables and $c_1 \ldots c_n$ are characters of a string constant. There are $n+1$ possible ways to split c over X and Y such that $X \cdot Y = c$: either $X = \epsilon, Y = c_1 c_2 \ldots c_n$, or $X = c_1, Y = c_2 c_3 \ldots c_n$, etc., or finally $X = c_1 c_2 \ldots c_n, Y = \epsilon$. Note that each of these equations represents a case that can be explored by the solver, and also that all of these cases are mutually exclusive (as clearly X cannot be equal to both ϵ and c_1 simultaneously, etc.). However, the Boolean abstraction constructed over theory literals hides the fact that these are mutually exclusive cases, and so it is reasonable to expect that the search performance can be improved by preventing the core solver from considering more than one of these cases simultaneously. A naïve solution is to encode $O(n^2)$ extra mutual exclusion Boolean clauses over these variables. This, of course, results in quadratic blowup in formula size and can result in very poor performance. Alternatively, we can do nothing and let the congruence closure solver in the Z3 core discover the mutual exclusivity of these Boolean variables on its own. However, this can result in unnecessary backtracking, unnecessary calls to congruence closure, and, in the worst case, reduces to the same set of mutual exclusion clauses being learned in the form of conflict clauses. It also does not actually prevent the core solver from exploring multiple mutually exclusive branches at least once, since it must do so in order to learn that these clauses cannot be true simultaneously.

The means of handling such cardinality constraints efficiently has been well-studied; previous work has investigated the possibility of alternate encodings, e.g. totalizers [11] and lazy cardinality [7]. The implementation in Z3str4, by contrast, shows a way to handle these constraints in the inner loop of the SAT solver in a theory-aware manner. This means that theory solvers do not have to perform rewriting or assert extra clauses to enforce mutual exclusivity of choices. Instead, they can provide this information directly to the core solver, which can use these facts during the search. This saves on the propagation effort of the DPLL(T) framework. My implementation of this technique is as follows:

1. The theory solver provides the core solver with a set S of mutually exclusive literals that correspond to a theory case-split. This set is maintained by the core solver in a list of all such sets.

2. During branching, the core solver checks if the current branching literal belongs to some such set S. If it does, the current branching literal is assigned true and all other theory case-split literals in S are assigned false. Otherwise, the default branching behaviour is used.

3. During propagation, the core solver may assign a truth value to a literal l in some set S of theory case-split literals. If so, the theory case-split check is invoked, i.e.,

the core solver checks whether two literals l_1, l_2 in the same set S have been assigned the value true. If this is the case, the core solver immediately generates the conflict clause $(\neg l_1 \lor \neg l_2)$.

4.7 Clause Sharing

As Z3str4 invokes various solvers in a selected arm, it may happen that one of the invoked algorithms either times out or fails to find a solution. For example, if the length abstraction solver reaches the maximum iteration count without either finding a satisfying assignment or deciding unsatisfiability, it will give up and return UNKNOWN. In a typical tactic-based SMT solving approach, the next tactic will be tried, or else UNKNOWN will be returned if no tactic is successful. However, this method has a weakness, in that each tactic is tried in isolation and from scratch. Any information learned by previous solvers, in the form of conflict clauses or facts derived from the input formula, is lost once the next algorithm is called. This can result in redundant effort being expended by solvers in checking branches of the search tree that have already been explored and found to be unsatisfiable by previous solvers. In the extreme worst case, every solver could get "stuck" exploring the same search space.

To address this, I implemented a mechanism in Z3's SMT architecture wherein a theory solver can indicate during the search that one or more constraints are to be shared to future solvers in the event that it fails. The length abstraction solver and arrangement solver both share blocked length assignments learned during the search with subsequent solvers. The requirement which must be met in order for this to remain sound and complete is that each shared constraint must be implied by the original input formula and cannot contain any new variables that do not appear in the original formula. Any SAT or UNSAT answer to this augmented formula is then equisatisfiable with the original formula.

Clause sharing has been implemented previously in parallelized satisfiability solvers such as ManySAT [35], and is most effective when small, general clauses can be shared. In this instance, the different arms of Z3str4 are invoked sequentially, but clause sharing is still beneficial since both the length abstraction solver and arrangement solver rely heavily on length information during the search. Thus, it may be possible that without clause sharing, both algorithms could check the same candidate length assignments during the search. If a combination of lengths was already shown to be unsatisfiable by a previous solver, blocking that assignment, at the very least, prevents subsequent algorithms from expending redundant effort in checking that assignment again.

4.8 Empirical Evaluation

In this section, I report on the overall performance of Z3str4 over 20 different benchmark suites obtained from industrial applications, fuzz testing, and solver developers. Detailed evaluation of the tool is crucial to understand the extent of any potential increase in performance and ability to solve string instances, and to compare it to existing tools to understand where it stands with respect to the state of the art. This allows me to validate the effectiveness of the techniques I have presented so far, and to demonstrate correctness (empirically).

4.8.1 Empirical Setup and Solvers Used

Z3str4 is compared against three other leading string solvers available today. CVC4 [44] is a general-purpose SMT solver which uses algebraic methods to reason about strings. Z3str3 [15] is the previous solver in the Z3-str family, and uses the arrangement reduction technique previously described here. Z3seq [62] is the Z3 sequence solver, implemented by Nikolaj Bjørner and others at Microsoft Research, as part of the Z3 theorem prover. This evaluation used CVC4's binary version 1.8, commit `59e9c87` of Z3str3, and the sequence solver included in Z3's binary version 4.8.9. In order to evaluate the effectiveness of Z3str4's probe technique, I also present results for two other configurations of probes and arms of Z3str4. The first configuration, labelled "Z3str4-1probe", only uses the final probe (for the conjunctive fragment) and arms illustrated in Figure 4.2. The second configuration, labelled "Z3str4-2probe", uses the final two probes (word equations and conjunctive fragment). The configuration labelled "Z3str4" uses all three probes.

I did not compare against the Z3str2 [73] or Norn [6] solvers as neither tool supports the full SMT-LIB version 2.6 standard. Z3str2 in particular is missing support for the `str.to_int` and `str.from_int` terms for string-integer conversion. Norn is missing support for many high-level string terms such as `str.indexof` or `str.substr` which are used in the benchmarks. The ABC [9] solver handles string and length constraints by conversion to automata. However, their method over-approximates the solution set of the input formula, which may be unsound. Thus, I excluded ABC from the evaluation. I was also unable to evaluate against Trau [5] as the provided source code did not compile. Finally, I did not evaluate against the OSTRICH [21] or Z3-Trau [3] solvers as they do not support the full SMT-LIB version 2.6 standard for strings, and additionally both of these tools encountered significant runtime issues during empirical evaluation of the regex fragment (as described later in Section 5.4).

All evaluations were performed on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory using the ZaligVinder [42] benchmarking framework. A 20 second timeout was used. The models generated by each solver for satisfiable instances were cross-validated against all competing solvers. If a competing solver reported that a claimed model was not a valid satisfying assignment to the input formula, this was recorded in the results as a soundness error. In the case where a solver reported "UNSAT" for an instance on which a competing solver found a valid satisfying assignment, the "UNSAT" answer was treated as incorrect and recorded as a soundness error. If at least one solver answered "UNSAT" for a given instance and no competing solvers answered "SAT", this was treated as a correct answer. This is necessary because the tools considered either do not produce proofs that can be checked, or the proofs they produce are incompatible with the input format expected by competing solvers. With respect to the results presented here, I was careful to conduct multiple runs and to cross-validate results both within and between runs. The presented figures are typical of the performance of the tools I evaluated over multiple runs. For a random single query, the sample variance in execution time for 100 independent runs was 0.001 (0.07% of average execution time). For the full set of benchmarks used, the variance is negligible.

4.8.2 Benchmarks Used

I evaluated the efficiency of Z3str4 and competing solvers over 20 different benchmark suites containing over 120000 instances and covering a wide range of applications and input terms. To the best of my knowledge, this is the largest and most comprehensive collection of string instances that has been used for testing string solvers to date. Most of these suites were obtained from industrial applications and solver developers.

Of the 20 suites I used to conduct the evaluation, 16 are from previously published sources, comprising over 70% of the total number of instances. The BanditFuzz suite was obtained via private communication with the authors. The Automatark, StringFuzz-regex-generated, and StringFuzz-regex-transformed benchmarks are regex-heavy benchmarks which I generated for the purpose of evaluating regular expression performance of Z3str4 and other SMT solvers; I describe these benchmarks (and evaluate them in isolation) in Section 5.4.2.

The following benchmark suites are derived from previously published applications: PyEx [58], SMTLIB25 [13], IBM PISA [65], Norn [6], Trau Light [4], Leetcode [4], IBM AppScan [1, 72], Sloth [36], Woorpje [24], Kaluza [59], StringFuzz [17], Z3str3 regression [15], Cashew [18], JOACO [66], Stranger [70], and Kausler [40, 2].

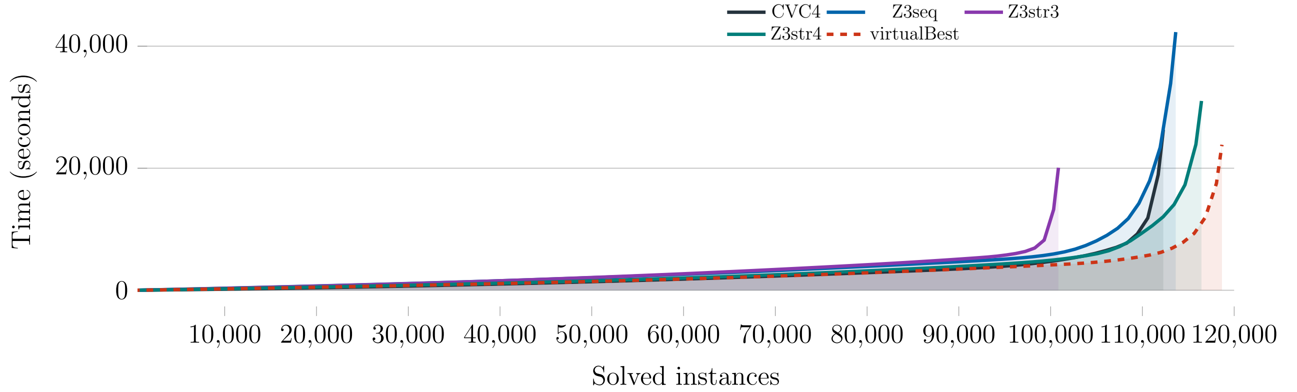

Figure 4.6: Cactus plot showing performance of string solvers on combined benchmarks.

	CVC4	Z3seq	Z3str3	Z3str4	Virtual Best
sat	68386	69853	59663	**71842**	74001
unsat	43897	43783	41198	**44597**	44659
unknown	**40**	50	949	170	454
timeout	8043	6680	18556	**3757**	1241
soundness error	**0**	**0**	9	**0**	0
program crashes	**0**	**0**	**0**	**0**	0
Total correct	112283	113636	100852	**116439**	118660
Time (s)	187941.021	176134.056	391379.159	**107456.964**	50993.467
Time w/o timeouts (s)	27080.148	42534.056	**20259.159**	32316.964	23840.060

Table 4.1: Table showing detailed results of string solvers on combined benchmarks.

4.8.3 Results and Analysis

A summary of the results is presented in Figure 4.6 and Table 4.1. The cactus plot in Figure 4.6 shows the cumulative time taken by each solver on all cases in increasing order of runtime. Solvers that are further to the right and closer to the bottom of the plot have better performance. A series for the "virtual best solver" is also shown, which is calculated by simulating perfect arm selection, that is, using the arm of Z3str4 with the best individual performance on each instance.

Overall, Z3str4 outperforms CVC4, Z3str3, and Z3seq, solving more instances and having a lower total solving time than every other solver and with no errors or crashes. Z3str4 solves 4156 more cases than CVC4, 2803 more cases than Z3seq, and 15587 more cases than Z3str3. Including timeouts, Z3str4 is 42.8% faster than CVC4, 39.0% faster than Z3seq, and 72.5% faster than Z3str3. Furthermore, Z3str4 approaches the virtual best solver in terms of number of queries solved (98.1%) and comes the closest in terms of total time taken (210.7%). The overall results indicate that Z3str4 is highly effective at solving a wide variety of practical string instances.

The experimental results make clear the effectiveness of using algorithm selection to solve a wide variety of string problems, and highlight the significant performance improvement obtained over using either Z3str3 or Z3seq alone. The results also highlight the advantages of using multiple static feature probes to perform algorithm selection.

4.8.4 Performance Analysis of Components of Z3str4

To evaluate the architecture of Z3str4 and better understand its component algorithms, I categorize the queries by the arm they are assigned to and compare these algorithms on the queries they are meant to do well on versus the queries they are not meant to do well on. Across the entire set of benchmarks used for evaluation, there are 35345 regex queries, 42522 higher-order queries, 15113 conjunctive queries, and 23643 non-conjunctive queries. I exclude 3743 queries that are solved by the simplifier.

LAS Performance Analysis

I hypothesize that LAS will do comparatively better in the conjunctive fragment because it will learn more general lessons and its underlying bit-vector solver will be quicker every iteration. Empirically this hypothesis holds: LAS solves comparatively more queries per second than the arrangement solver in the conjunctive fragment (1128.5%) than it does

	Z3str4-las	Z3str4-arr	Z3str4-seq
sat	11693	11887	12115
unsat	2479	2592	2532
unknown	941	90	1
timeout	0	544	465
soundness error	0	8	0
program crashes	0	0	0
Total correct	14172	14471	14647
Time (s)	1089.591	12555.877	10056.023
Time w/o timeouts (s)	1089.591	1675.877	756.023

Table 4.2: All three component solvers on queries in the conjunctive fragment.

outside the conjunctive fragment (904.6%). Overall, LAS is extremely effective in the conjunctive fragment, especially when used as the first solver in an arm.

To analyze the performance of LAS and the conjunctive fragment probe, I identify all queries that do not have regular expressions and do not mostly consist of word equations, divide these remaining benchmarks into two categories – conjunctive fragment and non-conjunctive fragment – and then compare the performance of all three component solvers on both sets.

Conjunctive Fragment. In the conjunctive fragment, there are 15113 queries. The full results are summarized in Table 4.2 and Figure 4.7.

Non-Conjunctive Fragment. In the non-conjunctive fragment, there are 23643 queries. The full results are summarized in Table 4.3 and Figure 4.8.

Arrangement Solver Performance Analysis

The arrangement solver with the bit-vector backend significantly improves performance over the arrangement solver without the bit-vector backend, both in terms of time and

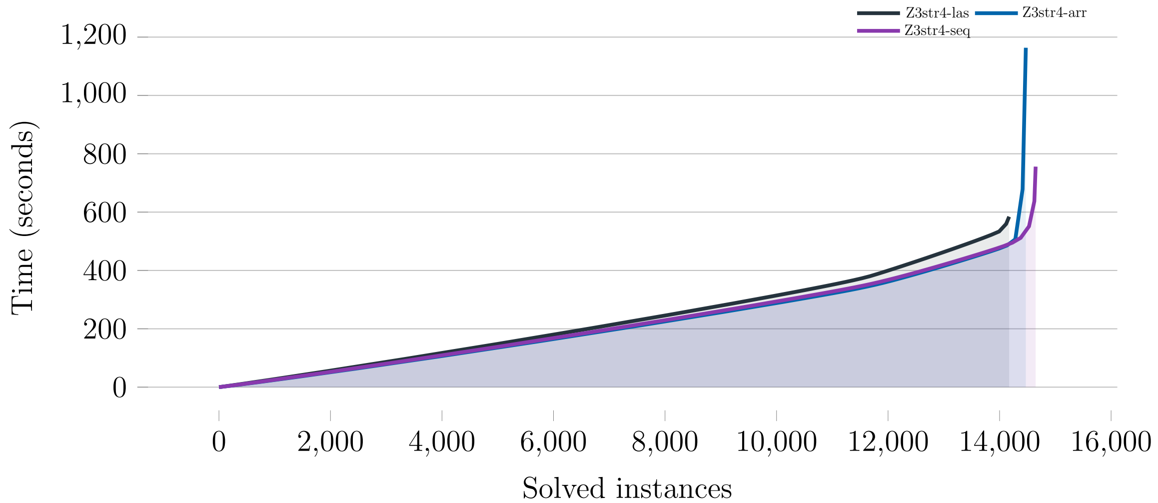

Figure 4.7: All three component solvers on queries in the conjunctive fragment.

number of instances solved. Without the bit-vector backend, the arrangement solver solves 72417 instances in 423949.163 seconds; with the bit-vector backend, the arrangement solver solves 107401 instances (148.3% of the queries without) in 262047.893 seconds (61.8% of the time without).

To analyze the performance of the bit-vector backend in the arrangement solver, I run the solver on all queries with the bit-vector backend enabled and disabled, and compare. Cumulative data is visualised in Table 4.4 and Figure 4.9.

Sequence Solver Performance Analysis

Empirically, the sequence solver is best when most constraints are not word equations. In this fragment, the sequence solver solves 39639 cases in 74565.085 seconds, while the next best solver, the arrangement solver, solves only 31376 (79.1%) in 220076.933 seconds (295.1% of the sequence solver's time).

To analyze the performance of the sequence solver and the word equations probe, I find all queries that do not have regular expressions and divide these into those which have mostly word equations and those that do not, then compare the performance of all three component solvers on both sets.

	Z3str4-las	Z3str4-arr	Z3str4-seq
sat	13478	13611	13740
unsat	8988	9509	9341
unknown	1177	51	1
timeout	0	472	561
soundness error	0	0	0
program crashes	0	0	0
Total correct	22466	23120	23081
Time (s)	1139.155	10604.341	12013.431
Time w/o timeouts (s)	1139.155	1164.341	793.431

Table 4.3: All three component solvers on queries outside the conjunctive fragment.

Few Word Equations. In the "few word equations" fragment, there are 42522 queries. The full results are summarized in Table 4.5 and Figure 4.10.

Many word Equations. In the "many word equations" fragment, there are 38756 queries. The full results are summarized in Table 4.6 and Figure 4.11.

Impact of Clause Sharing

Clause sharing significantly reduces the amount of time taken and slightly increases the number of solved instances. In particular, over all benchmarks, with clause sharing turned off, Z3str4 solves 15 fewer cases and takes 2201.790 more seconds (102% of the time taken with clause sharing).

To analyze the performance of clause sharing, I run the full Z3str4 solver on all queries with clause sharing enabled and disabled. Cumulative data is visualised in Table 4.7 and Figure 4.12.

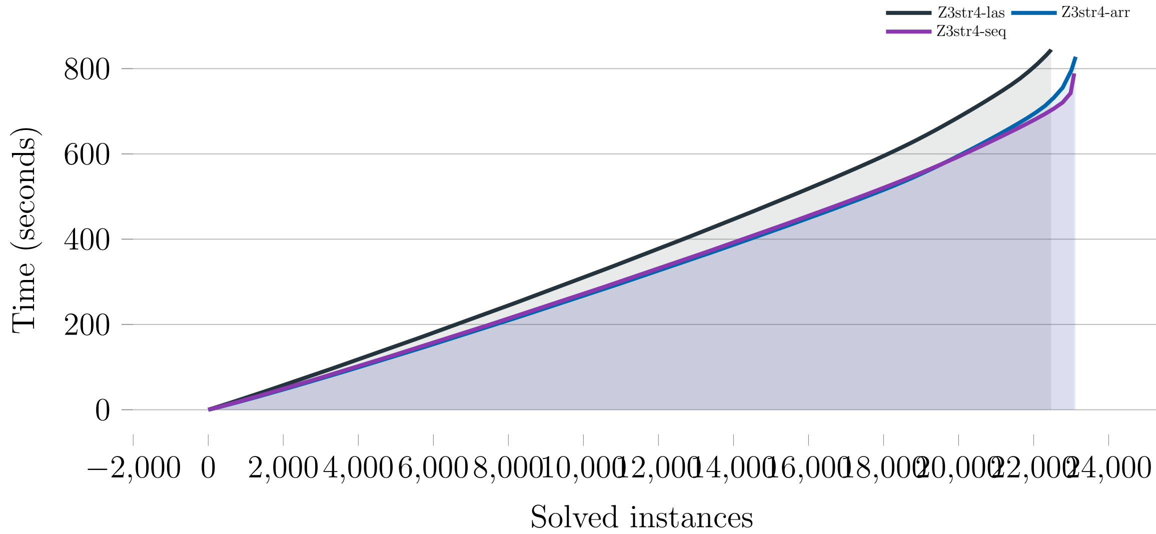

Figure 4.8: All three component solvers on queries outside the conjunctive fragment.

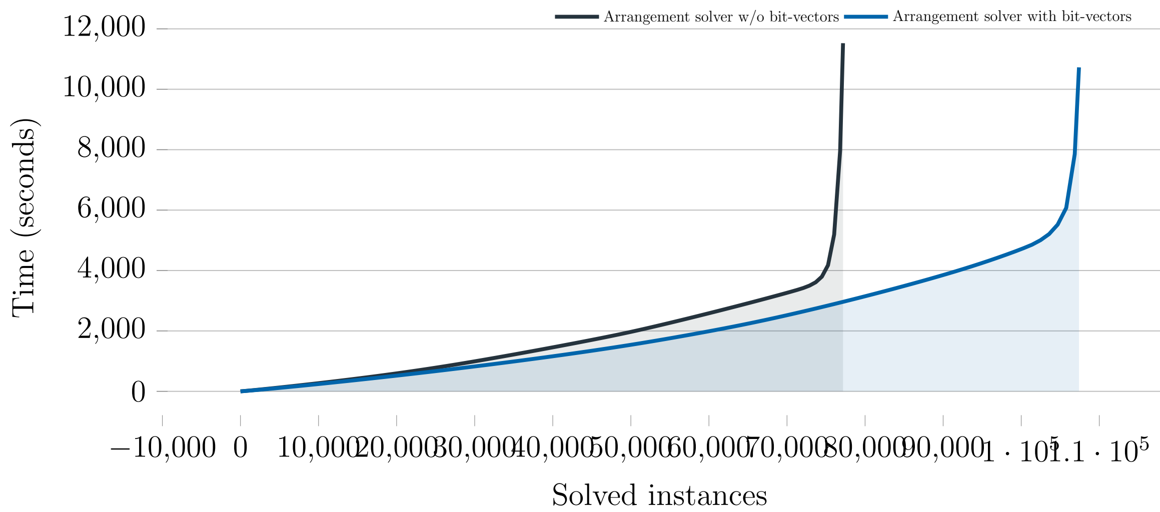

Figure 4.9: Cactus plot of string solvers on all benchmarks. Timeout=20 s. Timeout, unknown, and error instances excluded.

	Without BV Backend	With BV Backend
sat	65073	63095
unsat	29583	44314
unknown	5707	525
timeout	20003	12432
soundness error	22239	8
program crashes	3	0
Total correct	72417	107401
Time (s)	423949.163	262047.893
Time w/o timeouts (s)	23889.163	13407.893

Table 4.4: Cumulative Results. Timeout=20 s. Total time includes all solved, timeout, unknown and error instances.

	Z3str4-las	Z3str4-arr	Z3str4-seq
sat	13185	14937	23187
unsat	15805	16439	16452
unknown	13532	314	83
timeout	0	10832	2800
soundness error	0	0	0
program crashes	141	0	0
Total correct	28990	31376	39639
Time (s)	12088.081	220076.933	74565.085
Time w/o timeouts (s)	12088.081	3436.933	18565.085

Table 4.5: All three component solvers on queries in the “few word equations” fragment.

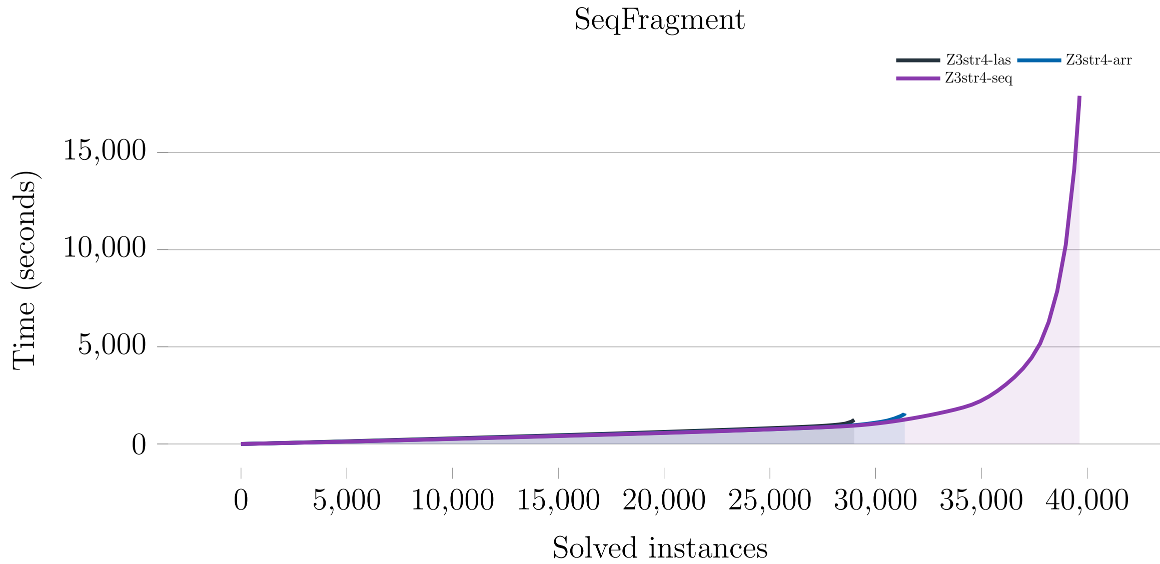

Figure 4.10: Performance of all three component solvers on queries in the "few word equations" fragment

	Z3str4-las	Z3str4-arr	Z3str4-seq
sat	25171	25498	25855
unsat	11467	12101	11873
unknown	2118	141	2
timeout	0	1016	1026
soundness error	0	8	0
program crashes	0	0	0
Total correct	36638	37591	37728
Time (s)	2228.746	23160.218	22069.454
Time w/o timeouts (s)	2228.746	2840.218	1549.454

Table 4.6: Performance of all three component solvers on queries in the "many word equations" fragment.

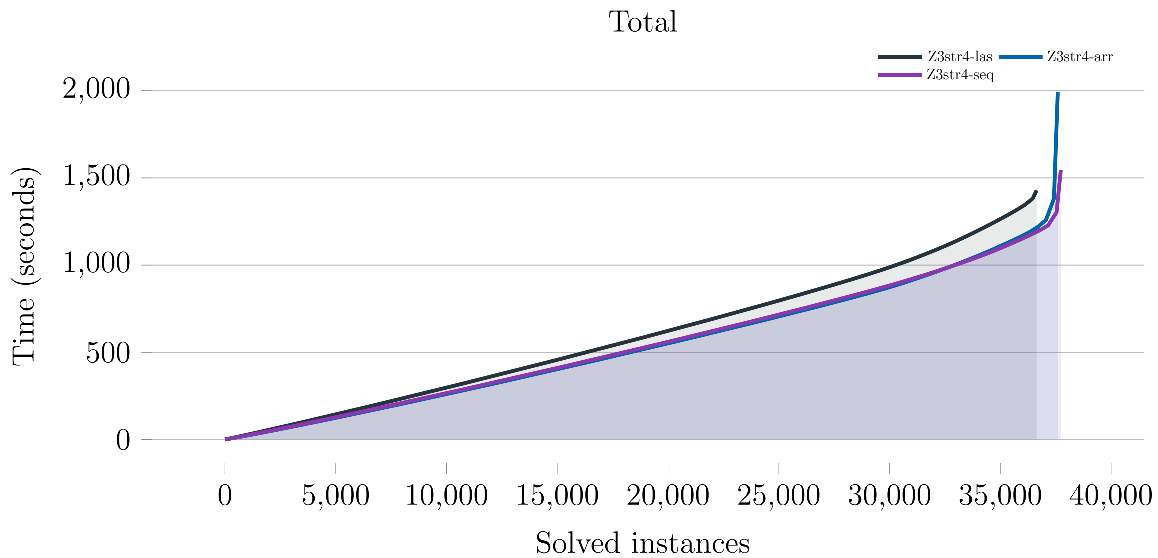

Figure 4.11: Performance of all three component solvers on queries in the "many word equations" fragment

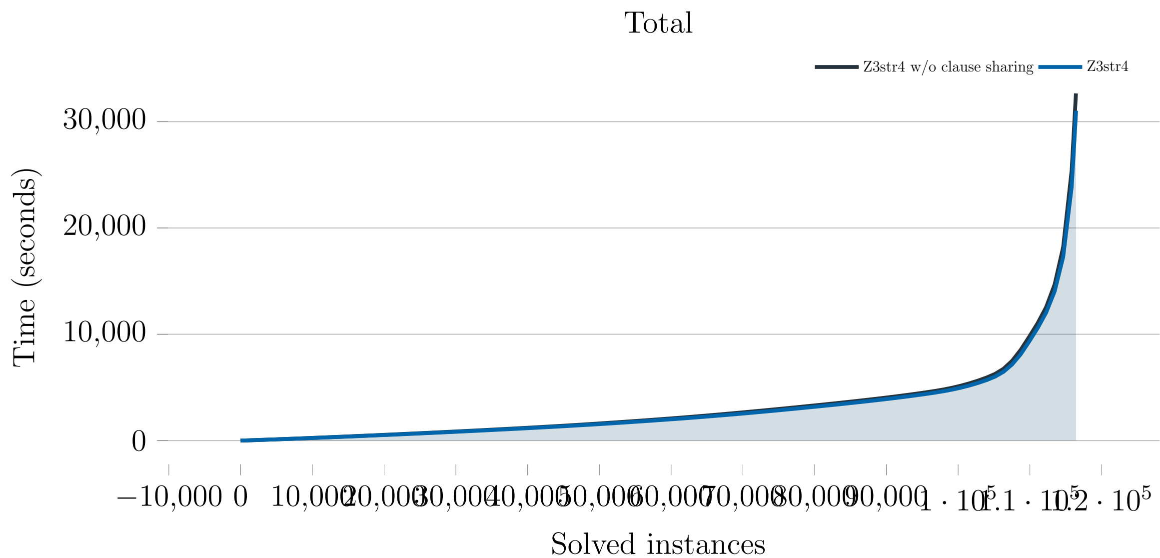

Figure 4.12: Cactus plot of Z3str4 on all benchmarks with clause sharing enabled/disabled. Timeout=20 s. Timeout, unknown, and error instances excluded.

	Clause sharing off	Clause sharing on
sat	71827	71842
unsat	44595	44597
unknown	159	170
timeout	3785	3757
soundness error	0	0
program crashes	1	0
Total correct	116422	116439
Time (s)	109658.754	107456.964
Time w/o timeouts (s)	33958.754	323169.64

Table 4.7: Cumulative results for Z3str4 with clause sharing enabled/disabled. Timeout=20 s. Total time includes all solved, timeout, unknown and error instances.

Chapter 5

Algorithms and Heuristics for Theories over Regular Expressions and Linear Arithmetic over String Length

In this chapter, I describe the algorithms and heuristics used by Z3str4 to solve constraints in the theory T_{LRE}, which include regular expression constraints, linear integer arithmetic constraints, and string length. In Section 5.1 I begin by explaining the motivation for new algorithms and heuristics to reason about this theory. I then present the main algorithm used by Z3str4 to solve the fragment T_{LRE} in Section 5.2. I describe a number of novel and general heuristics which Z3str4 uses to reason about these constraints more efficiently in Section 5.3. Finally, in Section 5.4, I demonstrate the power of this algorithm and these heuristics via an extensive empirical evaluation over a large and diverse benchmark of 57256 regex-heavy instances. Z3str4's regex solver outperforms five other state-of-the-art string solvers, namely CVC4, OSTRICH, Z3seq, Z3str3, and Z3-Trau over this benchmark.

5.1 Background and Motivation

SMT solvers that support theories over regular expression (regex) membership predicates and linear arithmetic over length of strings have enabled many important applications in the context of analysis of string-intensive programs. Examples include symbolic execution

Algorithm 5: The length-aware algorithm for the theory T_{LRE} of regex and integer constraints

```
   Input    : Conjunction φ of constraints of the form S ∈ RE, and conjunction ψ of linear integer arithmetic
              constraints
   Output   : SAT or UNSAT
1  forall constraints S ∈ RE in φ do
2  |  L_S ← ComputeLengthAbstraction(S) ;
3  |  L_RE ← ComputeLengthAbstraction(RE) ;
4  |  if ψ ∪ L_S ∪ L_RE inconsistent then
5  |  |  return UNSAT
6  |  end
7  |  refine L_S as tightly as possible with respect to L_RE;
8  end
9  forall strings S_i occurring in φ do
10 |  let R be the set of all regexes RE in all terms S_i ∈ RE ;
11 |  I ← intersection of all regular expressions in R ;
12 |  if I is empty then
13 |  |  return UNSAT
14 |  else
15 |  |  L_I ← ComputeLengthAbstraction(I) ;
16 |  end
17 end
18 𝓛_S ← the union of all length abstractions L_S;
19 𝓛_RE ← the union of all length abstractions L_RE;
20 𝓛_I ← the union of all length abstractions L_I;
21 if ψ ∪ 𝓛_S ∪ 𝓛_RE ∪ 𝓛_I has any solution M then
22 |  forall strings S occurring in φ do
23 |  |  obtain len(S) from M ;
24 |  |  let 𝒜 be the set of all automata for all regexes RE in all terms S ∈ RE ;
25 |  |  Automaton J ← intersection of all terms in 𝒜 ;
26 |  |  S ← any string of length len(S) in J ;
27 |  end
28 |  return SAT
29 else
30 |  return UNSAT
31 end
```

and path analysis [16, 57], as well as security analyzers that make use of string and regex constraints for input sanitization and validation [10, 59, 61]. Common to all these applications is the requirement for a rich quantifier-free (QF) first-order theory over strings, regexes, and integer arithmetic. Unfortunately, as shown in Chapter 3, the quantifier-free first-order theory of strings containing regex constraints, linear integer arithmetic on string length, string-number conversion, and string concatenation (even without word equations) is undecidable. Additionally, many non-trivial fragments of this theory containing regular expression constraints are hard to decide; they typically have exponential-space lower bounds or are PSPACE-complete. The task of creating efficient solvers and algorithms that handle practical constraints belonging to fragments of this theory remains a very difficult challenge.

Many modern solvers handle regex constraints via an automata-based approach. Such methods are powerful and intuitive, as regular expressions can be converted to deterministic or non-deterministic finite automata and automata-based representations and computations involving them such as intersection, emptiness, etc. are well-established. However, solvers must handle two key practical challenges in this context.

The first challenge is that many automata operations, such as intersection, are computationally expensive, yet handling these operations is required in order to solve constraints that are relevant to real-world applications. For example, given an input formula containing the constraints $X \in R_1 \land X \in R_2$, for a string term X and regex terms R_1, R_2, if the languages R_1 and R_2 have no common strings, then computing, or somehow reasoning about, the intersection $R_1 \cap R_2$ will be required for the solver to show unsatisfibility of these constraints. Deciding whether this intersection is empty is PSPACE-complete in general [63].

The second challenge relates to the integration of length information with regex constraints. The lengths of all possible strings accepted by an automaton can be represented as a finite union of arithmetic progressions[22, 34]. This implies a disjunction of linear length constraints that must be satisfied. These constraints are often more challenging for solvers to handle than a conjunction. Additionally, other "implied" length constraints often exist in practical instances. For example, an equality between two string terms $X = Y$ implies that X and Y have the same length. If the input formula further constrains $X \in R_1 \land Y \in R_2$, the solver can no longer reason about R_1 or R_2 in isolation as there is an additional constraint on the lengths of X and Y that limits the possible solutions that can be considered.

The challenges of using automata-based methods can be addressed via the use of *lazy extraction of implied length constraints* and *lazy regex heuristics* in order to avoid performing expensive automata operations when possible. In the remainder of this chapter, I introduce a length-aware automata-based algorithm for solving regex constraints and linear integer arithmetic over length of string terms, and several length-aware heuristics that enable efficient reasoning about practical regex constraints. I also present the results of an empirical evaluation of this algorithm and these heuristics on a large collection of randomized and industrial instances against other state-of-the-art SMT solvers.

5.2 Algorithm for Solving Regex, Length, and Linear Arithmetic Constraints

In this section, I present a novel decision procedure for the quantifier-free first-order theory T_{LRE} over regex membership predicate and linear integer arithmetic over string length.

The pseudocode presented in Algorithm 5 is shown at a high level that captures the essence of the procedure being performed. Implementation-specific details are omitted for clarity. Z3str4 incorporates a version of this algorithm as part of a DPLL(T) theory combination with Z3's core solver for Boolean formulas and arithmetic solver for integer arithmetic constraints. A high-level architectural view of this algorithm is also illustrated in Figure 5.1.

The algorithm takes as input a conjunction ϕ of regex membership constraints and a conjunction ψ of linear integer arithmetic constraints over the lengths of string variables appearing in ϕ. Without loss of generality, it is assumed that all constraints in ϕ are positive; negative constraints $S \notin RE$ can be replaced with the positive complement $S \in \overline{RE}$. The algorithm returns SAT if there is a satisfying assignment to all string variables consistent with the regex constraints ϕ and length constraints ψ. It is assumed that the algorithm has access to a procedure for checking the consistency of linear integer arithmetic constraints and for obtaining satisfying assignments to these constraints (in our implementation, this is fulfilled by Z3's arithmetic solver).

Lines 1–8 check whether the length information implied by ϕ is consistent with ψ. The function `ComputeLengthAbstraction` takes as input either a string term S or a regex RE and computes a system of length constraints corresponding to either an abstraction of derived length information from string constraints or an abstraction of length information derived from the regex RE. For example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. If the length abstractions are inconsistent with the given length constraints, there can be no solution which satisfies both the length and regex constraints, and hence the algorithm returns UNSAT. Otherwise, line 7 refines the length abstraction L_S with respect to the regex RE. This improves the efficiency of finding solutions to the augmented system of length constraints later in the algorithm. In our implementation, the lower and upper bounds of the length of S are checked against the lengths of accepting paths in the automaton for RE. For instance, if L_S implies that $len(S) \geq 5$, but the shortest accepting path in the automaton has length 7, the lower bound is refined to $len(S) \geq 7$.

Lines 9–17 check that the intersection of all automata constraining each string vari-

able is non-empty. Although intersecting automata is relatively expensive (as it runs in quadratic time w.r.t. the size of the intersected automata), it is still more efficient to do this before enumerating length assignments, and taking the intersection here is necessary to maintain soundness. (The heuristics in Section 5.3 illustrate some methods by which this computation can be made more efficient or even avoided.) If the length information is consistent, the algorithm adds a length abstraction constraint L_I encoding the lengths of all possible solutions to the intersection I.

By construction of $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$, the input formula is satisfiable iff this system of integer constraints has a solution. If such a solution M exists, lines 22–28 construct an assignment for each string variable with respect to its length assignment. A solution must exist as the lengths of strings considered are limited to those lengths for which the intersection of the corresponding automata is non-empty; the solution is consistent by construction with both the input length constraints and string constraints. If a solution M does not exist, then the constraints $\phi \wedge \psi$ are not jointly satisfiable, and the algorithm returns UNSAT.

I demonstrate soundness, completeness, and termination of Algorithm 5 as follows. On line 4 the algorithm checks whether $\psi \cup L_S \cup L_{RE}$ is satisfiable. If not, it returns UNSAT on line 5. Lines 9–17 check whether the intersection of regex constraints for each string variable is empty. If so, it returns UNSAT; otherwise, it adds an additional constraint encoding the lengths of all strings in this intersection. Therefore, $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$ has a solution iff there exists an assignment to each string variable that is consistent with the arithmetic constraints ψ and that corresponds to the length of a solution in the intersection of its regex constraints $\mathcal{L}_I$. Lines 22–28 construct this solution if it exists. Therefore, Algorithm 5 is a decision procedure for the QF first-order theory of regex constraints, string length, and linear integer arithmetic.

As previously mentioned, Z3str4 supports other high-level operations that are not part of this theory via existing support in the arrangement solver. An extension to this algorithm provides support for including these operations, which may render the theory undecidable. These terms are not in Algorithm 5 because their inclusion would make the algorithm incomplete (see Chapter 3). Algorithm 5 describes the part of the implementation which is novel and complete.

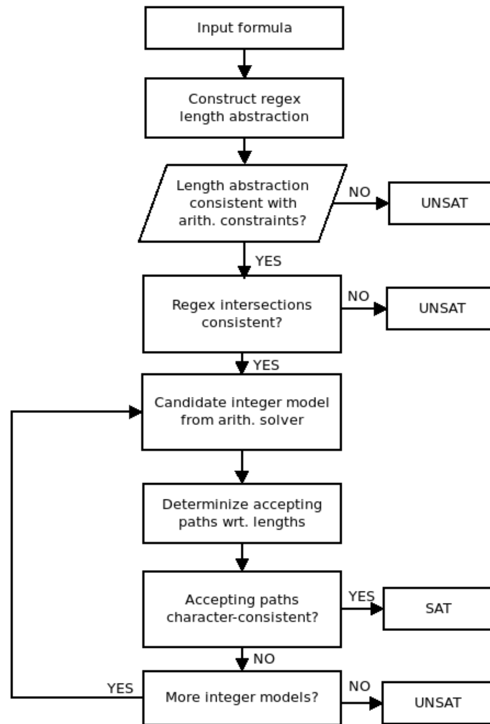

Figure 5.1: Architecture of Z3str4's regular expression solving algorithm.

5.3 Length-Aware Heuristics for Solving Regular Expression Constraints

In this section, I describe several length-aware heuristics used by Z3str4 to improve the efficiency of regular expression reasoning. The idea of length-aware heuristics was discussed in Section 4.5, where I illustrated the fixed-length model construction procedure used in Z3str4 and mentioned the improvements achieved by Z3str2 over Z3-str by using length information to guide the search in the string and integer theory solvers. With respect to regular expressions, similar approaches can be used to obtain length information from regex constraints and use existing length facts to determine whether regex constraints are satisfiable.

5.3.1 Computing Length Information from Regexes

The first length-aware heuristic used is when constructing the length abstraction on line 3. If the regex can be easily converted to a system of equations describing the lengths of all possible solutions (for instance, in the case when it does not contain any complements or intersections), this system can be returned as the abstraction without constructing the automaton for RE yet. As previously illustrated, for example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. Note that this can be done from the syntax of the regex without converting it to an automaton. Deriving length information from the automaton would be simple by, for example, constructing a corresponding unary automaton and converting to Chrobak normal form [22]. However, performing automata construction lazily means that the algorithm cannot rely on having an automaton in all cases; this technique also provides length information even when constructing an automaton would be expensive.

In cases where the length abstraction cannot be inferred directly, the heuristic will fix a lower bound on the length of words in RE, and possibly an upper bound if it exists. Reasoning about the length abstraction early in the procedure gives the algorithm the opportunity to detect inconsistencies before expensive automaton operations are performed. This gives the arithmetic solver more opportunities to propagate facts discovered by refinement and potentially more chances to find inconsistencies or learn further derived facts.

5.3.2 Optimizing Automata Operations via Length Information

Similarly, computing the intersection I in line 11 is done lazily in the implementation of Z3str4 and over several iterations of the algorithm. The most expensive intersection operations can be performed at the end of the search, after as much other information as possible has been learned. The following heuristics are applied recursively to estimate the "cost" of each operation without actually constructing any automata:

- For a string constant, the estimated cost is the length of the string.
- For a concatenation or a union of two regex terms X and Y, the estimated cost is the sum of the estimates for X and Y.
- For a regex term X^*, the estimated cost is twice the estimate for X.

- For a regex term X under complement, the estimated cost is the product of the estimates obtained from subterms of X.

In essence, the constructions which "blow up" the least are expected to be the least expensive and are performed first. In the best-case scenario, this could mean avoiding the most expensive operations completely if an intersection of smaller automata ends up being empty. In the worst case, all intersections are computed eventually, as this is necessary to maintain the soundness of this approach.

5.3.3 Leveraging Length Information to Optimize Search

The algorithm communicates integer assignments and lower/upper bounds with the external arithmetic solver in order to prune the search space. The search for length assignments is done in practice as an abstraction-refinement loop involving Z3's arithmetic solver – for instance, by the methods described in Section 4.5. The arithmetic solver proposes a single candidate model for the system of arithmetic constraints; the regex algorithm checks whether that model has a corresponding solution over the regex constraints. If it does not, it asserts a conflict clause blocking that combination of length assignments and regex constraints from being considered again. This is necessary in a CDCL(T)-style solver such as Z3 in order to handle Boolean structure in the input formula.

5.3.4 Constructing Over-Approximated Prefixes/Suffixes to Find Empty Intersections

As previously mentioned, computing automata intersections is expensive, but in many cases it is necessary in order to prove that a set of intersecting regex constraints has no solution. In some cases, this can be done "by inspection" from the syntax of the regex terms without constructing or intersecting any automata. From the structure of a regular expression, it is easy to determine the first letter of all possible accepted strings that it matches. If several regexes would be intersected over the same string term, this is used to check whether these regexes have a prefix of length one in common. If they do not, their intersection cannot contain any strings other than the empty string (the heuristic also checks whether the empty string could be accepted by a similar syntactic approach). A similar construction for suffixes of length 1 is also used. In this way, the heuristic can infer that the intersection of several regex constraints is either empty, resulting in a conflict clause, or can only contain the empty string, resulting in a new fact and a simplification of

the formula – without actually constructing the intersection or, in fact, constructing any automata for these regexes.

For example, consider the following regex constraints on a variable X:

$$X \in (abc)^*$$
$$X \in a^+ \mid b^+$$

The prefix/suffix heuristic would proceed as follows. In the first constraint, the pattern abc is matched zero or more times, and could be empty; therefore, either X is empty or it must start with a and end with c. In the second constraint, each pattern is matched at least once, and cannot be empty; therefore X must start with a or b, end with a or b, and cannot be the empty string. Observe that according to the prefix heuristic, these constraints are consistent, since a is a valid prefix of both regexes; however, according to the suffix heuristic, they are inconsistent, as the possible suffixes a and b of the second regex do not include c, and the empty string is not a solution to both constraints. Hence it follows that these constraints are not jointly satisfiable. At this point, the heuristic would assert a conflict clause.

As demonstrated, all of these facts are derived from the syntax of the regular expression; the heuristic does not need to construct an automaton for either constraint in order to reason about them. By constructing an over-approximation of the possible solutions of X allowed by regex constraints, the heuristic can determine that their intersection is empty (or can only contain the empty string) without computing it precisely (which, as previously mentioned, is expensive to do and also requires constructing automata first). The heuristic is limited to prefixes and suffixes of length 1 as each additional letter causes the space required to keep track of these prefixes/suffixes to increase exponentially.

As an observation, this heuristic is inspired partly by the work of Brzozowski on regex derivatives [19]. The actual heuristic introduced here is conceptually different as it examines possible prefixes (and suffixes) of strings that could be accepted by a regex in order to demonstrate unsatisfiability, rather than examining the set of all possible suffixes given a fixed prefix in order to demonstrate satisfiability. This heuristic computes suffixes as well, whereas Brzozowski derivatives are traditionally computed with respect to prefixes of a string. Newer versions of Z3seq, including the one used in the empirical evaluation, use an algorithm based on symbolic derivatives to reason about regular expressions [62].

5.4 Empirical Evaluation

In this section, I describe the empirical evaluation of the regular expression algorithm and heuristics presented in this chapter as part of Z3str4. The aim of this evaluation is too validate the effectiveness of the techniques presented, as well as the correctness and efficiency of the implementation against other solvers. Additionally, I evaluate different configurations of the tool in order to demonstrate the efficacy of the heuristics I present.

5.4.1 Empirical Setup and Solvers Used

Z3str4 is compared against five other leading string solvers available today. CVC4 [44] is a general-purpose SMT solver which reasons about strings and regular expressions algebraically. Z3str3 [15] is the previous solver in the Z3-str family, and uses a reduction to word equations to reason about regular expressions. Z3seq [62] is the Z3 sequence solver, implemented by Nikolaj Bjørner and others at Microsoft Research, as part of the Z3 theorem prover. Z3seq uses a new theory of derivatives for solving extended regular expressions. Z3-Trau [3] is also based on Z3 and uses an automata-based approach known as "flat automata" with both under- and over-approximations. OSTRICH [21] uses a reduction from string functions (including word equations) to a model-checking problem that is solved using the SLOTH tool and an implementation of IC3. This evaluation used CVC4's binary version 1.8, commit `59e9c87` of Z3str3, the sequence solver included in Z3's binary version 4.8.9, Z3-Trau commit `1628747`, and OSTRICH version 1.0.1. CVC4, Z3seq, Z3str3, and Z3str4 support the latest full SMT-LIB standard for strings. Z3-Trau and OSTRICH do not support the entire SMT-LIB version 2.6 standard for strings, but they do support enough of the fragment used in this set of benchmarks to be considered in the evaluation.

I did not compare against the Z3str2 [73] or Norn [6] solvers as neither tool supports the `str.to_int` or `str.from_int` terms which represent string-number conversion, which are used in some sanitizer benchmarks. Additionally, Norn does not support many of the other high-level string terms such as `indexof` or `substr` which are used in the benchmarks. The ABC [9] solver handles string and length constraints by conversion to automata. However, their method over-approximates the solution set of the input formula, which may be unsound. Thus, I excluded ABC from the evaluation. I was also unable to evaluate against Trau [5] as the provided source code did not compile. All evaluations were performed on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory using the ZaligVinder [42] benchmarking framework. A 20 second timeout was used. The models generated by each solver for satisfiable instances were cross-validated

against all competing solvers. If a competing solver reported that a claimed model was not a valid satisfying assignment to the input formula, this was recorded in the results as a soundness error. In the case where a solver reported "UNSAT" for an instance on which a competing solver found a valid satisfying assignment, the "UNSAT" answer was treated as incorrect and recorded as a soundness error. If at least one solver answered "UNSAT" for a given instance and no competing solvers answered "SAT", this was treated as a correct answer. This is necessary because the tools considered either do not produce proofs that can be checked, or the proofs they produce are incompatible with the input format expected by competing solvers.

Note that since the regular expression algorithm in Z3str4 is the central focus of this evaluation, other features of Z3str4 that are not part of the regex algorithm or heuristics, including algorithm selection and the length abstraction solver, are disabled, and Z3str4 is run in a mode that uses the arrangement solver only.

5.4.2 Benchmarks

The comparison was performed on four suites of regex-based benchmarks with a total of 57256 instances. In total, almost 75% of the instances in this evaluation came from previously published industrial benchmarks or other solver developers. In the following paragraphs, I briefly describe each benchmark's origin and composition.

AutomatArk is a set of 19979 benchmarks based on a collection of real-world regex queries collected by Loris D'Antoni from the University of Wisconsin, Madison, USA. I translated the provided regexes [23] into SMT-LIB syntax resulting in two sets of instances: a "simple" set with a single regex membership predicate per instance, and a "complex" set with 2–5 regex membership predicates (possibly negated) over a single variable per instance. The instances in this benchmark are evenly divided between simple and complex problems.

RegEx-Collected is a set of 22425 instances taken from existing benchmarks with the purpose of evaluating the performance of solvers against real-world regex instances. This benchmark includes all instances from the AppScan [72], BanditFuzz,[1] JOACO [66], Kaluza [59], Norn [6], Sloth [36], Stranger [70], and Z3str3-regression [15] benchmarks in which at least one regex membership constraint appears.[2] No additional restrictions are placed on which

[1]The BanditFuzz benchmark is an unpublished suite obtained via private communication with the authors.

[2]Other benchmark suites available to me, including the PyEx, PISA, and Kausler benchmarks, did not include any regex membership constraints.

instances were chosen besides the presence of at least one regex membership predicate. I chose to evaluate against this benchmark in order to test the performance of solvers against instances that are already known to be challenging to solve and that have appeared in previously published and widely distributed benchmark suites. Additionally, these instances may contain regex terms in any context and with any other supported string operators. As a result, the benchmark is also exemplary of how string solvers perform in the presence of operations and predicates that are relevant to program analysis.

StringFuzz-regex-generated is a set of 4170 problems generated by the StringFuzz string instance fuzzing tool [17]. These instances only contain regular expression and linear arithmetic constraints. The motivation in choosing this benchmark is to isolate and evaluate the regex performance of a string solver in the context of mixed regex and arithmetic constraints. Tools with better regex and arithmetic solvers should perform better. Fuzz testing, as performed in the **StringFuzz-regex-generated** benchmark, has been shown to be extremely productive in discovering bugs and performance issues in SMT solvers. I chose to include these instances because they enable us to isolate the performance of the solver on regex-heavy constraints in a way that the industrial benchmarks or instances obtained from other solver developers cannot.

StringFuzz-regex-transformed is a set of 10682 instances which were produced by transforming existing industrial instances with StringFuzz. To create the **StringFuzz-regex-transformed** benchmark, I applied StringFuzz's transformers to instances supplied by Amazon Web Services related to security policy validation, handcrafted instances which are inspired by real-world input validation vulnerabilities, and the regex test cases included in Z3str3's regression test suite. The instances in this suite include regex constraints, arithmetic constraints on string length, string-number conversion (*numstr*), string concatenation, word equations, and other high-level string operations such as `charAt`, `indexof`, and `substr`. As is typical for fuzzing in software testing, the goal is to create a suite of tests from a given input that are similar in structure but that explore interesting behaviour not captured by a "typical" industrial instance. These transformed instances are often harder than the original industrial ones.

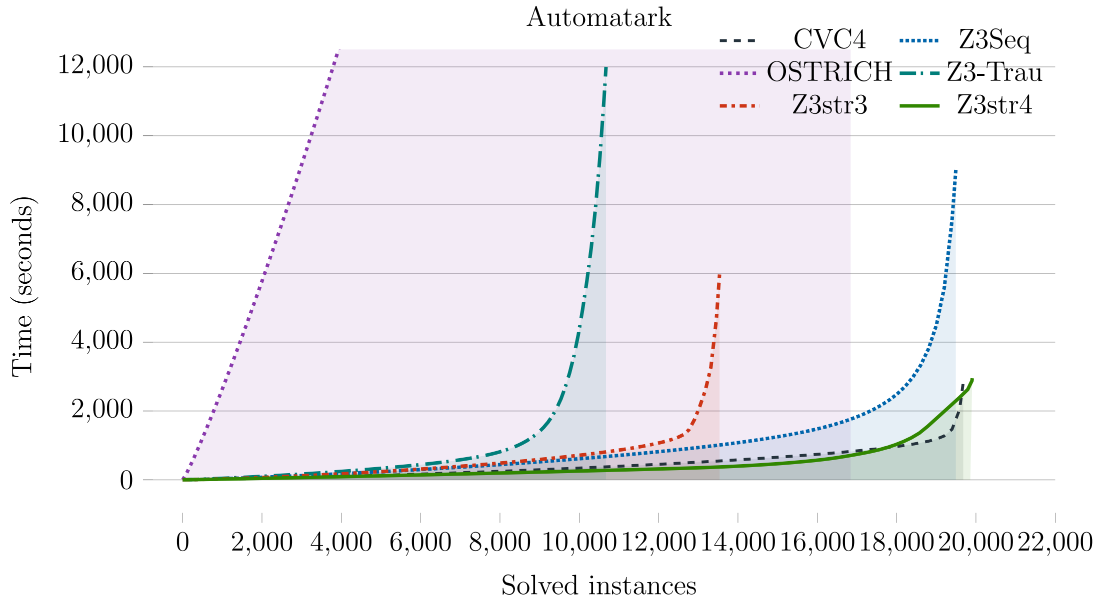

Figure 5.2: Cactus plot summarizing detailed performance on Automatark benchmark.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str4
sat	14376	14204	11461	8157	9151	**14459**
unsat	5304	5290	5381	3817	4385	**5450**
unknown	1	**0**	15	5045	406	5
timeout	298	485	3122	2960	6037	**69**
soundness error	**0**	**0**	**0**	1300	**0**	**0**
program crashes	**0**	**0**	**0**	1063	2	**0**
Total correct	19680	19494	16842	10674	13536	**19905**
Contribution score	1.0	1.0	**2.0**	–	0.0	0.5
Time (s)	8789.425	18718.425	158910.126	80021.352	126825.967	**4331.419**
Time w/o timeouts (s)	2829.425	9018.425	96470.126	20821.352	6085.967	**2951.419**

Table 5.1: Detailed results for the Automatark benchmark. Z3str4 has the biggest lead with a score of 1.01.

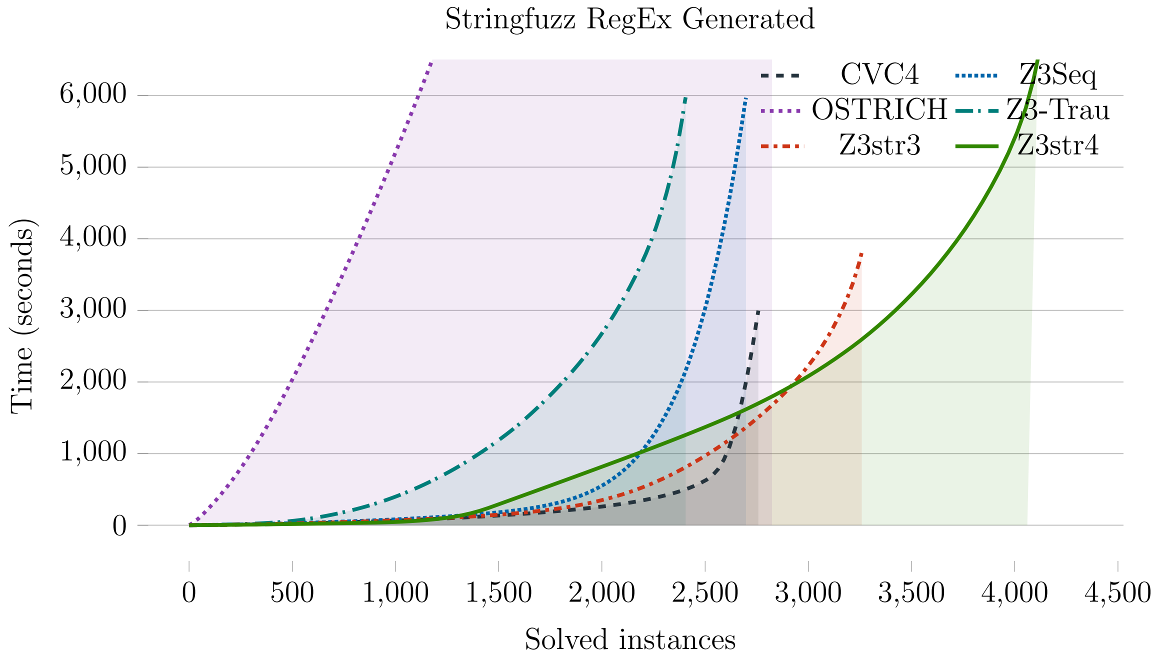

Figure 5.3: Cactus plot showing detailed results for the StringFuzz-regex-generated benchmark.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str4
sat	2316	2001	2005	1590	3227	**3282**
unsat	442	697	819	824	32	**830**
unknown	**0**	**0**	1	192	**0**	7
timeout	1412	1472	1345	1564	911	**51**
soundness error	**0**	**0**	**0**	8	**0**	**0**
program crashes	**0**	**0**	**0**	192	**0**	**0**
Total correct	2758	2698	2824	2406	3259	**4112**
Contribution score	0.0	**3.17**	2.0	–	0.0	0.17
Time (s)	31236.207	35409.000	51571.800	37323.550	22031.636	**7563.818**
Time w/o timeouts (s)	2996.207	5969.000	24671.800	6043.550	3811.636	**6543.818**

Table 5.2: Detailed results for the StringFuzz-regex-generated benchmark. Z3str4 has the biggest lead with a score of 1.25.

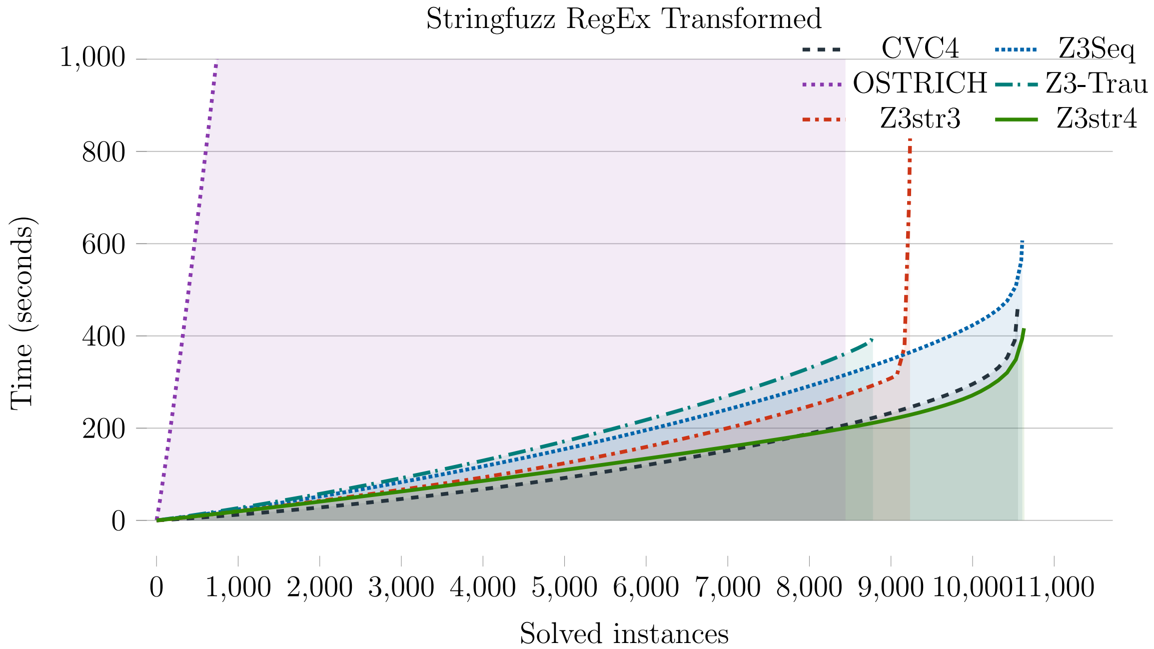

Figure 5.4: Cactus plot showing detailed results for the StringFuzz-regex-transformed benchmark.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str4
sat	4541	**4633**	3899	3672	4417	4617
unsat	6016	5976	4549	**6282**	4817	6062
unknown	**0**	**0**	2233	721	**0**	64
timeout	125	73	**1**	7	1448	3
soundness error	**0**	**0**	5	1241	**0**	**0**
program crashes	**0**	**0**	**0**	718	**0**	**0**
Total correct	10557	10609	8443	8713	9234	**10615**
Contribution score	0.5	0.0	–	–	0.0	**4.83**
Time (s)	2969.643	2066.935	23094.737	722.545	29788.245	**479.585**
Time w/o timeouts (s)	469.643	606.935	23074.737	582.545	828.245	**419.585**

Table 5.3: Detailed results for the StringFuzz-regex-transformed benchmark. Z3str4 has the biggest lead with a score of 1.0.

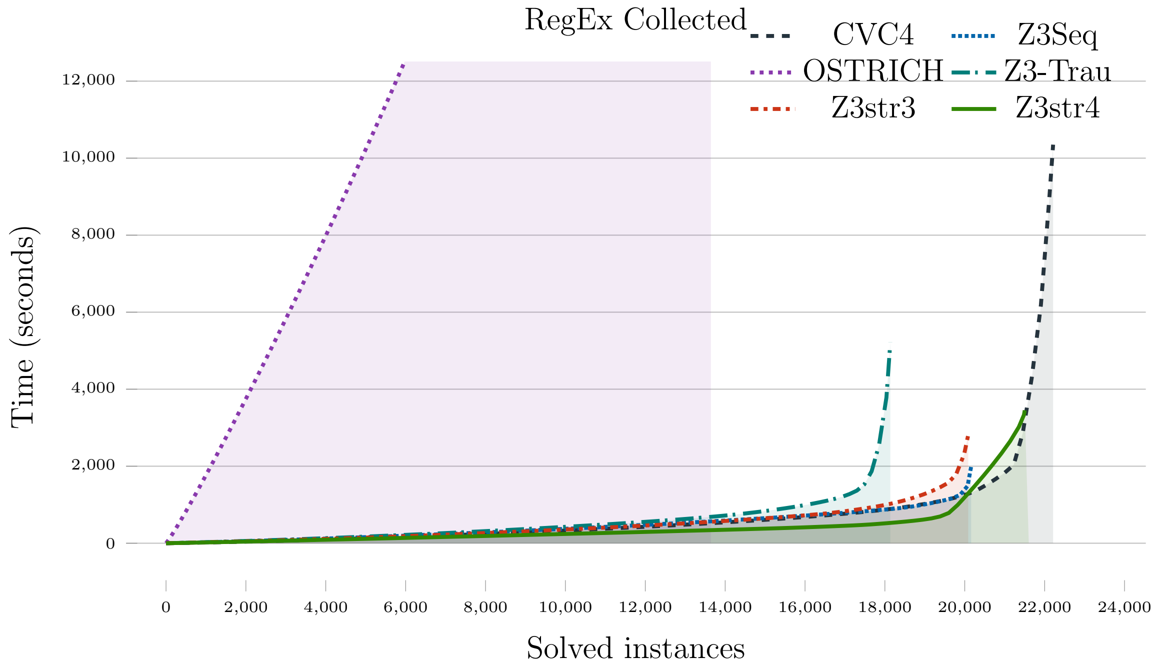

Figure 5.5: Cactus plot showing detailed performance for the RegEx-collected benchmark.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str4
sat	**12077**	10712	5134	10714	10768	11358
unsat	**10135**	9448	8532	10115	9332	10146
unknown	**0**	**0**	8652	546	758	125
timeout	213	2265	**107**	1050	1567	799
soundness error	**0**	**0**	23	2776	13	**0**
program crashes	**0**	**0**	**0**	504	**0**	**0**
Total correct	**22212**	20160	13643	18053	20087	21502
Contribution score	**91.06**	3.51	–	–	–	14.54
Time (s)	14610.224	47293.484	71666.750	32220.939	35053.106	20314.643
Time w/o timeouts (s)	10350.224	**1993.484**	69526.750	11220.939	3713.106	4334.643

Table 5.4: Detailed results for the RegEx-collected benchmark. CVC4 has the biggest lead with a score of 1.03.

5.4.3 Comparison and Scoring Methods

Solvers are compared directly against the total number of correctly solved cases, total time with and without timeouts, and total number of soundness errors and program crashes.

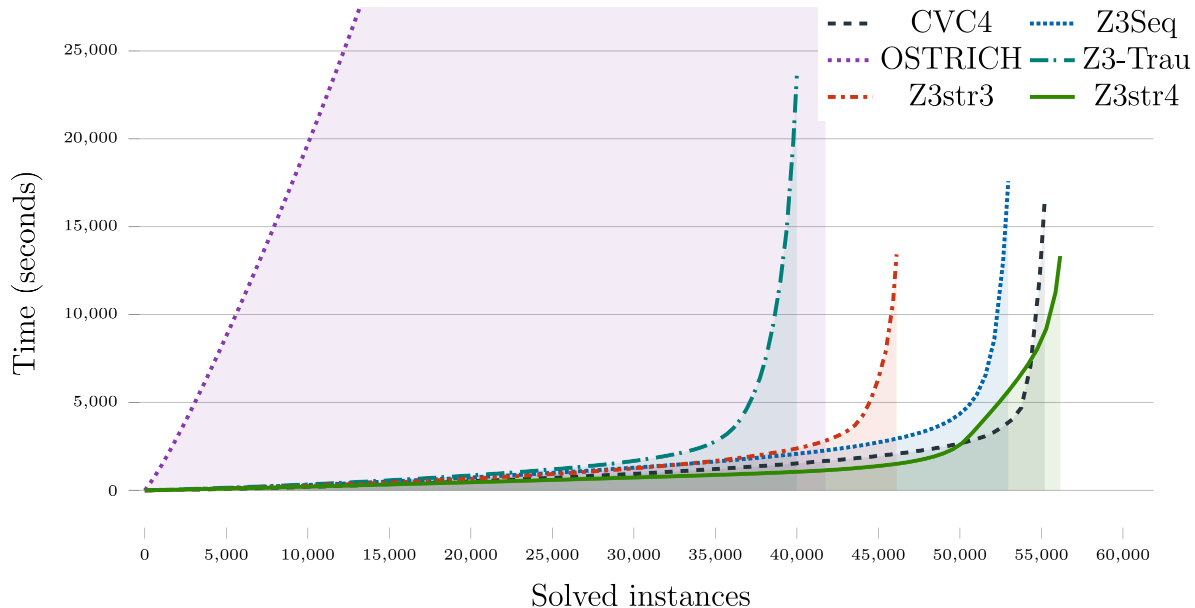

Figure 5.6: Cactus plot summarizing performance on all benchmarks. Z3str4 has the best overall performance.

I also computed the biggest lead winner and largest contribution ranking following the scoring system used by the SMT Competition [12]. Briefly, the biggest lead measures the proportion of correct answers of the leading tool to correct answers of the next ranking tool, and the contribution score measures what proportion of instances were solved the fastest by that solver. In accordance with the SMT Competition guidelines, a solver receives no contribution score (denoted as –) if it produces any incorrect answers on a given benchmark. In both cases, higher scores are better.

5.4.4 Analysis of Empirical Results

The cactus plot in Figure 5.6 shows the cumulative time taken by each solver on all cases in increasing order of runtime. Solvers that are further to the right and closer to the bottom of the plot have better performance.

Overall Z3str4 solves more instances and performs better than all competing solvers. Across all benchmarks, Z3str4 is over 1.7x faster than CVC4, 3.2x faster than Z3seq, 4.6x faster than Z3-Trau, 6.5x faster than Z3str3, and 9.3x faster than OSTRICH (including timeouts). Additionally, Z3str4 has fewer combined timeouts and unknowns than other

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str4
sat	33310	31550	22499	24133	27563	**33716**
unsat	21897	21411	19281	21038	18566	**22488**
unknown	**0**	**0**	10901	6504	1164	201
timeout	2049	4295	4575	5581	9963	**922**
soundness error	**0**	**0**	28	5325	13	**0**
program crashes	**0**	**0**	**0**	2477	2	**0**
Total correct	55207	52961	41752	39846	46116	**56134**
Contribution score	95.99	19.87	–	–	–	**145.07**
Time (s)	57625.499	103487.844	305243.413	150288.386	213698.954	**32689.465**
Time w/o timeouts (s)	16645.499	17587.844	213743.413	38668.386	14438.954	**14249.465**

Table 5.5: Combined results of string solvers on all benchmarks. **Z3str4** has the best overall performance on all benchmarks compared to CVC4, OSTRICH, Z3seq, Z3str3, and Z3-trau and the biggest lead with a score of 1.02.

tools considered, and no soundness errors or crashes. These results are summarized in Table 5.5. Notably, both Z3-Trau [3] and OSTRICH [21] had significant runtime issues in these experiments. Z3-Trau produced 5325 soundness errors and 2477 crashes on the benchmarks (13% of all instances), which is significantly higher than other tools used. OSTRICH produced 10901 “unknown” responses on the benchmarks (19% of all instances), due to both unsupported features and crashes, and also produced 28 soundness errors. According to SMT Competition scoring, Z3str4 won the division across all benchmarks with a lead of 1.02, and had the largest contribution to the division with a score of 145.07. CVC4 had a contribution score of 95.99, and Z3seq had a score of 19.87. OSTRICH, Z3-Trau, and Z3str3 received no contribution score as they each returned at least one incorrect answer.

The empirical results make clear the efficacy of length-aware automata-based techniques for regular expression constraints when accompanied with length constraints (which is typical for industrial instances). The effectiveness of the techniques illustrated in this chapter is demonstrated particularly by comparing Z3str4 with Z3str3. With the extra features of Z3str4 disabled as described above, the only differences between these tools are the length-aware regex algorithm and heuristics implemented in Z3str4 and bug fixes. By improving the regex algorithm and applying new heuristics, Z3str4 achieved a speedup of over 9x and solved over 10000 more cases than Z3str3.

5.4.5 Detailed Experimental Results

Figure 5.2 and Table 5.1 show the detailed results for the **AutomatArk** benchmark. In this benchmark, Z3str4 solves more instances than all other solvers, has the fewest timeouts/unknowns, and has the fastest overall running time. Including timeouts, Z3str4 is 2.2x faster than CVC4, 4.7x faster than Z3seq, 40.4x faster than OSTRICH, 20.4x faster than Z3-Trau, and 32.3x faster than Z3str3.

Figure 5.3 and Table 5.2 show the detailed results for the **StringFuzz-regex-generated** benchmark. Z3str4 solves more instances than all other solvers, has over 90% fewer timeouts than other solvers, no unknowns, and has the fastest overall running time. Including timeouts, Z3str4 is 6.1x faster than CVC4, 6.9x faster than Z3seq, 10x faster than OSTRICH, 7.3x faster than Z3-Trau, and 4.3x faster than Z3str3.

Figure 5.4 and Table 5.3 show the detailed results for the **StringFuzz-regex-transformed** benchmark. Z3str4 solves more instances in total than all other solvers and has the lowest total running time without timeouts. Including timeouts, Z3str4 is 2.7x faster than CVC4, 1.9x faster than Z3seq, 21x faster than OSTRICH, and 27x faster than Z3str3. Although Z3-Trau is 1.5x faster than Z3str4 on this benchmark, including timeouts, Z3-Trau also produces 1241 answers with soundness errors and crashes on 718 other cases. Z3str4 produces no wrong answers or soundness errors on the benchmark. Z3-Trau also solves 1923 fewer cases correctly in total than Z3str4.

Figure 5.5 and Table 5.4 show the detailed results for the **RegEx-Collected** benchmark. Z3str4 outperforms Z3seq, Z3str3, OSTRICH, and Z3-Trau on this benchmark and is competitive with CVC4 both in terms of total number of instances correctly solved and total running time. CVC4 solves 609 more instances than Z3str4 on this benchmark, but Z3str4 is 1.1x faster overall (including timeouts). Z3str4 is 3.6x faster than Z3seq, 5.4x faster than OSTRICH, 2.4x faster than Z3-Trau, and 2.6x faster than Z3str3.

5.4.6 Analysis of Individual Heuristics and Results

To demonstrate the effectiveness of individual heuristics described in Section 5.3 and implemented in Z3str4, I evaluated different configurations of the tool in which one or more heuristics were disabled. Figure 5.7 and Table 5.6 show the results. The plot line "Z3str3RE" shows the baseline performance of the tool with all heuristics enabled. The plot line "All heuristics off" shows the performance with all heuristics disabled. Each other series shows the performance with the named heuristic disabled and all others enabled. From the plots and table, it is clear that Z3str4 performs best with all heuristics

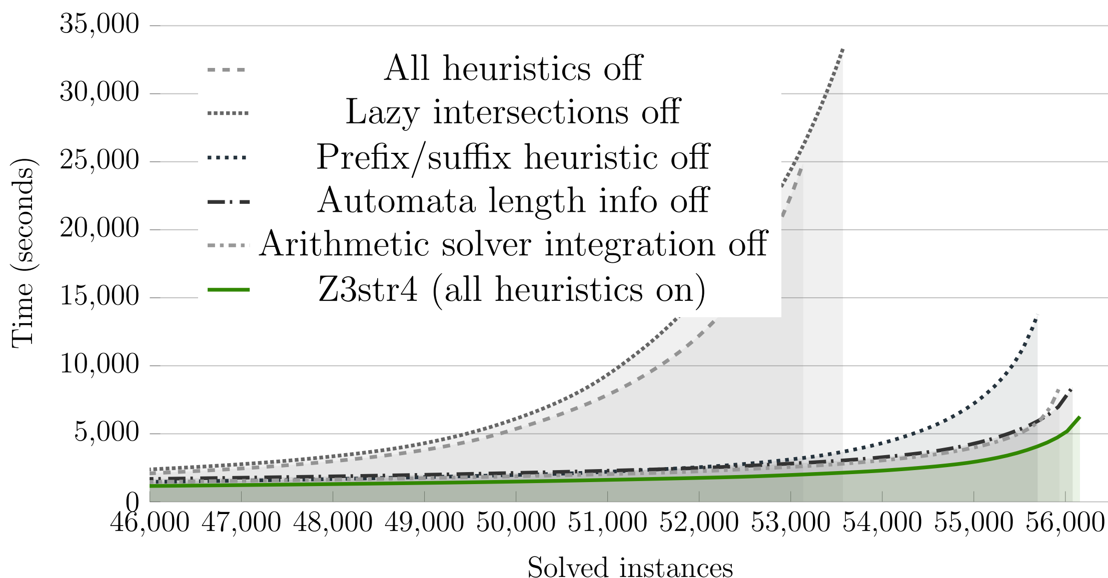

Figure 5.7: Cactus plot comparing performance by disabling individual heuristics on all benchmarks.

	All off	Lazy intersection off	Prefix/suffix off	Automata length info off	Arith. solver integ. off	Z3str4
sat	31046	31486	33817	33816	33804	**33820**
unsat	22090	22085	21880	22264	22131	**22339**
unknown	313	323	287	285	**283**	291
timeout	3807	3362	1272	891	1038	**806**
soundness error	**0**	**0**	**0**	**0**	**0**	**0**
program crashes	42	39	**0**	1	**0**	**0**
Total correct	53136	53571	55697	56080	55935	**56159**
Time (s)	102102.388	101799.263	40068.501	27178.746	30006.857	**23339.266**
Time w/o timeouts (s)	25962.388	34559.263	1462.8501	9358.746	9246.857	**7219.266**

Table 5.6: Comparison of different regular expression heuristics in Z3str4 on all benchmarks.

enabled and is 4.4x faster than using none. Every other configuration performs significantly worse relative to having all heuristics enabled. These results demonstrate empirically that every heuristic increases total solved instances and lowers total solver runtime, and that all heuristics can be used simultaneously for maximum efficacy.

Chapter 6

Observations and Future Work

In this chapter, I describe some opportunities for future theoretical and practical work related to the topics I have discussed in this thesis, and conclude with a summarization of the contributions I have presented.

6.1 Future Work

6.1.1 Theoretical Results

The results I have presented in this thesis come closer to an understanding of the decidability of the quantifier-free theory of word equations with length constraints, although they do not address it directly. It may be the case that new methods of reasoning or formulations are required to show decidability or undecidability of this theory.

Two additional restrictions of theories over strings with regular expressions are relevant: the quantifier-free theory of word equations with regular expression constraints, string-number conversion, and concatenation – but not length – and the quantifier-free theory of word equations with regular expression constraints, string length, and string-number conversion. The decidability of these theories remains open, even though (as shown in this thesis and in previous work) (un)decidability of directly stronger/weaker theories is known.

6.1.2 Algorithm Selection

The natural extension of the arm selection method described in Chapter 4 is a *machine learning based algorithm selection*. This brings to mind systems such as SATZilla [69], which uses a portfolio-based algorithm selection strategy that trains a machine learning model on features of a SAT formula in order to predict the performance of several different standalone solvers. Such features are fairly well understood for SAT formulas, but have not been explored in the broader context of SMT. An *empirical hardness model* for string formulas would be highly valuable both practically in developing predictive models for SMT solvers, and theoretically in understanding what makes certain string instances easy or hard to solve.

A more powerful "probe" or machine learning based algorithm selection strategy would also allow more algorithm combinations (arms) to be tested. Currently, Z3str4 is limited to static arms consisting of a fixed sequence of algorithms to be applied. Although the chosen arms have generally good performance, there may be other combinations that perform better than these arms in special cases. Understanding what combinations perform well and when/how to select them remains open for future development.

As well, Z3str4 uses a number of fixed parameters to different algorithms that are called, including timeout durations, maximum iteration upper bounds, etc. An interesting avenue of exploration is allowing these parameters to be tuned either offline via a machine-learning approach, or online via a dynamic adjustment algorithm.

6.1.3 Solving Regular Expression Constraints

As previously mentioned, newer versions of Z3's sequence solver use a regular expression reasoning procedure based on symbolic Boolean derivatives of regular expressions [62]. This has resulted in a significant performance increase for Z3seq compared to the automata-based algorithm that was previously used. It stands to reason whether Z3str4's regex solver could be improved in a similar way. The key challenges would include adapting the current automata-based heuristics, including lazy intersections, to a derivative-based reasoning method, and preserving the ability of the algorithm to extract length information without constructing automata.

The current length-aware heuristics implemented in Z3str4's regex solver are limited to a few operators. Future work could include extending these to more expressive functions and predicates, including `str.indexof`, `str.substr`, `str.to_int`, and `str.from_int`. Additionally, more advanced automata-based constructions could be used to extract a more

accurate length abstraction from a regex constraint. The benefits of this must be measured carefully against the overhead of constructing automata in more complex cases.

6.1.4 Theory-Aware Heuristics

The process of learning general length conflicts from the fixed-length model construction procedure described in Section 4.5 combined with the theory-aware branching and theory-aware case split heuristics suggests a more powerful idea of *theory-aware conflict clause learning* for string solvers. Currently, when the Boolean core solver in an SMT solver detects an inconsistency, it constructs a conflict clause to block the current partial assignment (and possibly others). In essence, a theory-aware conflict clause would be introduced by a theory solver when an inconsistency is detected between theory terms. Similar to how theory-aware branching provides theory-specific information to the core solver that it is otherwise unaware of, theory-aware conflict clause learning could potentially prune the search space in ways that are not possible to do with the Boolean abstraction alone. This could be done following rule-based solving procedures, for example [26].

6.1.5 Applications of String Solvers

String solvers are currently used in several program analysis and symbolic execution engines, and I anticipate that the need for more powerful solvers will only continue to grow over time as program analysis tools become better able to handle larger programs. I am especially interested in investigating integrations between strings and other theories, and examining whether string solvers should be extended with support for additional operators to provide better support for the way these operators are used in programming languages and applications. The integration of strings with bit-vector length, instead of integer length, was explored in a previous version of Z3str2 [64] and handling of strings in this way facilitates many low-level program analysis and exploit synthesis applications.

Currently, Z3str4 imposes the restriction that regex terms must be grounded, that is, they cannot contain variables. However, the SMT-LIB standard does permit string variables to appear in regex terms by way of the `str.to_re` operator. Reasoning about these so-called "symbolic regular expressions" presents a host of new challenges. Perhaps chief among them is the fact that a symbolic regex can no longer be so easily converted to an automaton. However, a novel combination of length-aware heuristics and reductions that use all available facts about string variables appearing in such regexes may be able to tackle this problem.

6.2 Conclusion

In this thesis, I have outlined numerous theoretical and practical contributions to the domain of string constraint solving. I demonstrated several theoretical results related to decidability and undecidability of theories over strings. I described an improved model construction procedure with a bit-vector backend that combines a word-based and unfolding-based approach for improved efficiency. I then presented the Z3str4 string solver, described its architecture and components, and illustrated a detailed empirical evaluation showcasing its performance against other state-of-the-art tools on a large and diverse set of benchmarks. Finally, I highlighted Z3str4's regular expression solving algorithm and several heuristics I implemented to make reasoning about regular expression constraints more efficient and to take advantage of length information in order to guide the search. I also quantified the performance of these improvements with in-depth empirical evaluations.

The contributions I have made are valuable from both a theoretical and practical standpoint. Of course, theoretical understanding of string solvers, decidability and undecidability, and algorithmic constructions are all important to improving the depth and breadth of our knowledge about the field. In addition, the practical contributions I have described in this thesis are available to the public as open source software, and are part of an industry-standard tool in the Z3 theorem prover. Applications of string solvers drive innovation and improvement in both the theory and the practice of string solving. More powerful string solvers and advancements in the state of the art in turn enable new applications that were not previously feasible. Indeed, industrial use cases and application-based benchmarks have motivated many of the contributions I have described here. I hope that users continue to benefit from Z3str4 in the future and take it beyond the limits of existing applications.

References

[1] IBM Security AppScan Tool and Source. URL: http://www-03.ibm.com/software/products/en/appscan-source.

[2] Kausler Suite.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 943–957, 2020.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. Trau: Smt solver for string constraints. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: A framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 602–617, New York, NY, USA, 2017. ACM.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.

[7] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. *To Encode or to Propagate? The Best Choice for Each Constraint in SAT*, pages 97–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] Roberto Amadini. A survey on string constraint solving, 2020.

[9] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272, Cham, 2015. Springer International Publishing.

[10] J. Backes, P. Bolignano, B. Cook, A. Gacek, K. S. Luckow, N. Rungta, M. Schaef, C. Schlesinger, R. Tanash, C. Varming, and M. Whalen. One-click formal methods. *IEEE Software*, 36(6):61–65, 2019.

[11] Olivier Bailleux and Yacine Boufkhad. *Efficient CNF Encoding of Boolean Cardinality Constraints*, pages 108–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[12] Haniel Barbosa, Jochen Hoenicke, and Antti Hyvarinen. 15th international satisfiability modulo theories competition (smt-comp 2020): Rules and procedures.

[13] Clark Barrett, Pascal Fontaine, Aina Niemetz, Mathias Preiner, and Hans-Jörg Schurr. Smt-lib benchmarks.

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[15] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59, 2017.

[16] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS ’09, pages 307–321, 2009.

[17] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. StringFuzz: A fuzzer for string solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 45–51. Springer, 2018.

[18] Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. Constraint normalization and parameterized caching for quantitative program analysis. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 535–546. ACM, 2017.

[19] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[20] J.Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 671–683. Springer New York, 1990.

[21] Taolue Chen, Matthew Hague, Anthony W Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[22] Marek Chrobak. Finite automata and unary languages. *Theor. Comput. Sci.*, 47(3):149–158, 1986.

[23] Loris D'Antoni. Automatark automata benchmark, 2018.

[24] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen. On solving word equations using SAT. In *Proc. RP*, volume 11674 of *LNCS*, pages 93–106. Springer, 2019.

[25] Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. The satisfiability of word equations: Decidable and undecidable theories. In Igor Potapov and Pierre-Alain Reynier, editors, *Reachability Problems - 12th International Conference, RP 2018, Proceedings*, volume 11123 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2018.

[26] Joel D. Day, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. Rule-based word equation solving. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, FormaliSE '20, page 87–97, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and*

algorithms for the construction and analysis of systems, TACAS'08, pages 337–340, 2008.

[28] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[29] Samuel Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.

[30] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[31] Amit Erez and Alexander Nadel. Finding Bounded Path in Graph Using SMT for Automatic Clock Routing. In *Proceedings of the 27th International Conference on Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer International Publishing, 2015.

[32] Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016.

[33] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: what's decidable? In *HVC'12*, 2012.

[34] Pawel Gawrychowski. Chrobak normal form revisited, with applications. In Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, and Denis Maurel, editors, *Implementation and Application of Automata - 16th International Conference, CIAA 2011, Blois, France, July 13-16, 2011. Proceedings*, volume 6807 of *Lecture Notes in Computer Science*, pages 142–153. Springer, 2011.

[35] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *J. Satisf. Boolean Model. Comput.*, 6(4):245–262, 2009.

[36] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.

[37] Artur Jeż. Recompression: Word equations and beyond. In *Developments in Language Theory*, Lecture Notes in Computer Science, pages 12–26. 2013.

[38] Artur Jeż. Recompression: Technique for word equations and compressed data. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications*, pages 44–67, Cham, 2020. Springer International Publishing.

[39] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. The expressibility of languages and relations by word equations. *J. ACM*, 47(3):483–505, May 2000.

[40] Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 259–270, New York, NY, USA, 2014. ACM.

[41] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.

[42] Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. The power of string solving: Simplicity of comparison. In *2020 IEEE/ACM 1st International Conference on Automation of Software Test (AST)*, pages 85–88. IEEE/ACM, 2020.

[43] F. W. Levi. On semigroups. *Bull. Calcutta Math. Soc.*, 36:141–146, 1944.

[44] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. Springer-Verlag, 2014.

[45] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. A decision procedure for regular membership and length constraints over unbounded strings. In *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, pages 135–150, 2015.

[46] Anthony W. Lin and Rupak Majumdar. Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 352–369. Springer, 2018.

[47] Anthony Widjaja Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 123–136. ACM, 2016.

[48] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English transl. in Math USSR Sbornik 32 (1977).

[49] Yu. Matiyasevich. Word equations, fibonacci numbers, and Hilbert's tenth problem. In *Workshop on Fibonacci Words*, 2007.

[50] Federico Mora. Private communication, 2021.

[51] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[52] Alexander Nadel. Routing under constraints. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 125–132, Austin, TX, 2016. FMCAD Inc.

[53] Rohit J Parikh. Language generating devices. *Quarterly Progress Report*, 60:199–212, 1961.

[54] Wojciech Plandowski. Satisfiability of word equations with constants is in pspace. *J. ACM*, 51(3):483–496, May 2004.

[55] Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, STOC '06, pages 467–476, 2006.

[56] Preiner, Mathias, Niemetz, Aina, Scott, Joseph, and Ganesh, Vijay. Machsmt: A machine learning-based algorithm selector for smt solvers. 2020.

[57] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 139–148, 2012.

[58] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli. Scaling up dpll (t) string solvers using context-dependent simplification. In *Proc. CAV*, pages 453–474. Springer, 2017.

[59] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[60] K. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer Berlin / Heidelberg, 1992.

[61] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[62] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. Technical Report MSR-TR-2020-25, Microsoft, August 2020.

[63] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time(preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, page 1–9, New York, NY, USA, 1973. Association for Computing Machinery.

[64] Sanu Subramanian. Bit-vector Support in Z3-str2 Solver and Automated Exploit Synthesis. Master's thesis, University of Waterloo, 2015.

[65] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, October 2013.

[66] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. *IEEE TSE*, 2018.

[67] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, 2014.

[68] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K.S. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007.

[69] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[70] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *Proc. TACAS*, TACAS'10, pages 154–157. Springer, 2010.

[71] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: an automata-based string analysis tool for php. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157, 2010.

[72] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, pages 1–40, 2016.

[73] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 235–254, 2015.

[74] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 114–124, New York, NY, USA, 2013. Association for Computing Machinery.