Fall 2014

Automated reasoning over string constraints

Tianyi Liang
University of Iowa

AUTOMATED REASONING OVER STRING CONSTRAINTS

by

Tianyi Liang

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2014

Thesis Supervisor: Professor Cesare Tinelli

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Tianyi Liang

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2014 graduation.

Thesis Committee: _____
Cesare Tinelli, Thesis Supervisor

_____
Aaron Stump

_____
Hantao Zhang

_____
Octav Chipara

_____
Clark Barrett

To Shihui Liang, Ji'er Chen, and Xue Mei

ACKNOWLEDGEMENTS

First and foremost I offer my sincerest gratitude to my advisor, Prof. Cesare Tinelli, for his continuous support throughout my entire PhD life and for his patience and knowledge. Without his guidance and inspiration, I would have never made it this far. I would like to thank him for leading me to the field of automated reasoning, for offering help whenever I needed it, and for all his sharp insight and passion.

Next, I would like to thank the other members of my dissertation committee for their advice and support. Prof. Aaron Stump is one of the kindest and most intelligent people I have ever met. He invited me to participate in his projects and taught me how to think deeply. His classes helped me understand sophisticated terminologies. Prof. Hantao Zhang has been generous and hospitable. He helped me to settle down in Iowa City, invited me to meet his family and advised me about academics and daily life. Prof. Octav Chipara is full of energy and always passionate about research. Exchanging ideas with him has inspired me, and more importantly his optimistic personality has pushed me forward. Prof. Clark Barrett is always fascinated with new ideas, especially for CVC4. This is a reason why CVC4 is the leading tool in the field. I really enjoy the good working atmosphere he maintains around him. I would like to thank him for inviting me to become part of this big CVC4 family.

Special thanks go to Dr. Andrew Reynolds (EPFL), Dr. Nestan Tsiskaridze, and Dr. Morgan Deters (NYU). We held regular meetings to discuss all issues in this work. This work would not have been possible without them. Dr. Reynolds helped

me with the initial implementation by extending his work on CVC4 equality engine. He deciphered the CVC4 architecture, shared his new ideas, as well as discussed algorithms. Dr. Tsiskaridze and I held intensive meetings for technical discussions. We went over every piece of proofs in this thesis, and she sent me her comments and corrections. I very much appreciate her insightful comments. Dr. Deters helped me with understanding CVC4 and running experiments on the clusters. When I encounter a problem, he is always ready to lend me a hand, whether it is lunch time or 2AM in the morning. Working with them has been an honor. I really value their contributions.

I would like to thank all the other members of the Computational Logic Center at the University of Iowa: John Bodeen, Adrien Champion, Harley Eades, Peng Fu, Pierre-Loïc Garoche, Temesghen Kahsai, Garrin Kimmell, Baoluo Meng, Ryan McCleeary, Duckki Oe, Md Hasib Shakur, Christoph Sticksel, and Ruoyu Zhang. I would also like to thank all the other members of CVC4: Kshitij Bansal (NYU), François Bobot (CEA), Chris Conway (Google), Liana Hadarean (NYU), Dejan Jovanovič (SRI), and Tim King (NYU). I have learned a lot from each one of them.

Most importantly, I would like to thank my parents, Shihui Liang and Ji'er Chen, for their complete selflessness and support as I have pursued my degree in the United States. As my parents tell me, they are my most reliable harbor, I can always sail myself there.

Finally, I would like to express my gratitude to my lovely wife, Xue Mei. I am very grateful for every encouragement and for every moment of happiness she has brought to me. Hand in hand, step by step, we will make ourselves a better life.

ABSTRACT

More and more applications in verification and security rely on automatic solvers, which can check the satisfiability of constraints over a rich set of data types, including character strings. Unfortunately, most string solvers today are standalone tools that can reason only about some fragment of the theory of strings and regular expressions, sometimes with strong restrictions on the expressiveness of their input language (such as, length bounds on all string variables). These specialized solvers reduce string problems to satisfiability problems over specific data types, such as bit vectors, or to automata decision problems. On the other hand, despite their power and success as back-end reasoning engines, general-purpose Satisfiability Modulo Theories (SMT) solvers so far have provided minimal or no native support for string reasoning.

This thesis presents a deductive calculus describing a new algebraic approach that allows solving constraints over the theory of unbounded strings and regular expressions natively, without reduction to other problems. We provide proofs of refutation soundness and solution soundness of our calculus, and solution completeness under a *fair* proof strategy. Moreover, we show that our calculus is a decision procedure for solving regular membership and length constraints.

We have implemented our calculus as a string solver for the theory of unbounded strings with concatenation, length, and membership in regular languages, and incorporated it into the SMT solver CVC4 to expand its already large set of built-

in theories. This work makes CVC4 the first SMT solver that is able to accept and process a rich set of mixed constraints over strings, integers, reals, arrays, and other data types. In addition, our initial experimental results show that, over string problems, CVC4 is highly competitive with specialized string solvers with a comparable input language. We believe that the approach we described in this thesis provides a new idea for string-based formal methods.

PUBLIC ABSTRACT

Privacy violation, or even impersonation, is one of the major concerns about online services, such as email systems or online shopping services. Techniques for vulnerability detection in such web-based applications provide a more secure environment for Internet services. A common approach for vulnerability detection formally describes a potential risk as a mathematical formula and reasons about this formula automatically. However, traditional mathematical methods focus more on numbers than strings, while the latter is considered as the main information carrier in web-based communication. This problem has been intensively studied, yet there is no existing method that can fully handle it.

We present a novel procedure for finding a solution to a formula containing strings, and prove correctness of our procedure. We have identified some patterns of formulas for which our procedure is guaranteed to terminate with a correct answer. In addition, our approach interacts with other procedures in a common framework, which allows it to find a solution to formulas containing not only strings but also multiple data structures.

We have implemented our approach in our solver CVC4 as an internal procedure, which makes it even more competitive as a general-purpose solver for formulas generated by vulnerability detection tools. We give a comparison with the state-of-the-art string solvers over benchmarks from real applications. Initial experimental results indicate that our approach is superior in terms of correctness and performance.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

This dissertation focuses on automated reasoning over string constraints. In particular, we present a set of algebraic approaches for solving constraints over the theory of unbounded strings and (extended) regular expressions natively, without reduction to other problems. With the integration into the DPLL(T) architecture, these approaches enable a general, multi-theory Satisfiability Modulo Theory (SMT) solver to reason about constraints combining strings, regular expressions, and other data types.

Ever since the invention of the first computer, software stability and reliability have been two key issues in the protection of privacy and security. As the desire for new functionalities in computer systems grows, the size of software application is exploding. It is obvious that one can either understand a small portion in a fairly large modern software system in detail or have a relatively abstract view of the whole system. The difficulty of fully understanding software in detail introduces more barriers in manual analysis, and thus puts the software robustness in danger. Meanwhile, the number of software vulnerabilities increases steadily. They bring inconvenience to users, cause billions of dollars in financial loss[1], and furthermore compromise national security.

[1]Hurricane Andrew (the most expensive natural disaster in the history of US) caused $25 billion dollars of damage, while the cost of the Love Bug virus was estimated to be between $3 billion to $15 billion [94].

With the migration from local applications to web-based applications, the amount of information sharing and service clouding is increased via networking. As a result, both industry and academia are paying more and more attention to the security of web applications. For example, the Open Web Application Security Project (OWASP) is defining standards for web application security. In particular, they have been updating their reports on major web application risks since 2004. In their 2013 report [67], the top five threats of web-based applications are: Injection, Broken Authentication, Cross-Site Scripting (XSS), Insecure Direct Object References, and Misconfiguration. Analysis of software will reduce the cost for the security issues.

In the last few years, a number of techniques originally developed for verification purposes have been adapted to support software security analyses as well. These techniques benefit from the rise of powerful specialized reasoning engines such as Satisfiability Module Theories (SMT) solvers. Security analyses frequently require the ability to reason about string values. One reason is that program inputs, especially in web-based applications, are often provided as strings which are then processed using operations such as matching against regular expressions, concatenation, and substring extraction or replacement. In general, both safety and security analyses could benefit from solvers that can check the satisfiability of constraints over a rich set of data types that includes character strings. Despite their power and success as back-end reasoning engines, however, general multi-theory SMT solvers so far have provided minimal or no native support for reasoning over strings.

1.1 Challenges and Motivation

A major difficulty of reasoning over strings is that the satisfiability problem of any reasonably comprehensive theory of character strings is undecidable [17]. However, the existential problem of several more restricted, but still quite useful, theories of strings is decidable. These include any theories of fixed-length strings, which are trivially decidable for having a finite domain, but also some fragments over unbounded strings (e.g., word equations [60]). Recent research on the string theory has focused on identifying decidable fragments suitable for program analysis [1] and, more crucially, on developing efficient solvers for them [89]. Unfortunately, most string solvers today are standalone tools that can reason only about (some fragment of) the theory of strings and regular expressions, sometimes with strong restrictions on the expressiveness of their input language such as, for instance, the imposition of exact length bounds on all string variables. These solvers are based on reductions to satisfiability problems over other data types, such as bit vectors, or to decision problems over automata.

Motivating Example

One successful security analysis method (symbolic execution) is to reduce some security problems to constraint satisfaction problems in some formal logic. In Figure 1.1, on the left side is a typical sample source code from a web application with several vulnerabilities, e.g., XSS [2] and buffer-overflow attacks. The middle code shows the corresponding constraints in SMT-LIB [13] format to verify whether the

[2]Users can execute a code from an outside website.

```
array a=explode('|',p);
string b=a[0];
int c=str2int(a[1]);
string y=c<strlen(b)?↩
    strncpy(y,b+c,strlen(b)↩
    -c):"";
output="<input type='text' ↩
    value='" + y + "'>";
```

```
(assert (= p (str.++ a0 "|" a1)))
(assert (= b a0))
(assert (= c (str.to.int a1)))
(assert (= y
  (ite (< c (str.len b))
  (str.substr b c (- (str.len b) c))
  "")))
(assert (str.contains y "http://"))
```

```
'><script>document↩
    .location='http↩
    ://attacker/?'+↩
    document.cookie↩
    </script><'|0"
```

Figure 1.1. A vulnerable code, its verification code, and one sample attack.

source code contains an XSS attack [3]. If the constraints are unsatisfiable, the source code is free of an XSS exploit; otherwise, there is an assignment (of variables in the constraints) that satisfies these constraints and defines a possible XSS attack.

The web-based application (on the left side of Figure 1.1) uses untrusted data, and then constructs an HTML form. One possible attack can use an input string (illustrated on the right side of Figure 1.1) to steal user authentication information, e.g., session ID. This information can be further used for unauthorized activities, e.g., impersonation. For example, many email systems keep users' geographic location information. Using a code similar to the example, an attacker may obtain a user session ID, and then use this information to locate the victim physically. The presence of this vulnerability can be detected by solving the constraints in the middle of Figure 1.1.

The success of exploiting vulnerabilities depends largely on the efficiency of constraint solving. In recent years, constraint solvers are extensively used in static [19, 20, 25] and dynamic analysis [31, 39] for vulnerability detection. However, traditional analysis tools use their own built-in constraint solvers. They require developers to

[3]For simplicity, the SMT-LIB constraints reflect only the XSS attack and do not include buffer-overflow check, as well as, variable definitions.

have a full knowledge of automated reasoning over all relevant logic [82]. Another choice is to encode a security analysis problem into a Satisfiability Modulo Theories (SMT) problem [13]. This method allows the analysis to directly use a pre-existing SMT solver, which combines a SAT solver with multiple specialized *theory solvers*. Especially with the advent of DPLL(T) framework [66] in recent years, the performance of SMT solvers has been dramatically improved. Thus, an SMT solver becomes a natural choice as the underlying constraint solver for a security tool.

1.2 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is about deciding the satisfiability of a formula in (decidable fragments of) first-order logic with respect to some *background theories*. A background theory fixes the interpretation of certain predicates and function symbols [12] so that the semantics of expressions is restricted to a set of models that belong to the theory.

Example 1 Consider the string function symbols con and len, the regular expression symbol star, the arithmetic function symbol $+$, the predicates $<$ and in, the string variables x, y, z, the constants a, b, and the expression $x = \mathsf{con}(y, z) \wedge (\mathsf{in}(y, \mathsf{star}(a)) \vee \mathsf{in}(z, \mathsf{star}(b))) \wedge \mathsf{len}(y) < \mathsf{len}(z)$ under the standard semantics. □

To find a solution for Example 1 by axiomatizing into first-order formulas may lead to the exponential growth of clause size, and thus become unsolvable in practice. In addition, it is hard to guarantee that every theory can be axiomatized. Meanwhile, a dedicated string solver cannot easily handle logical structure, e.g., disjunctions.

To overcome these difficulties, most state-of-the-art SMT solvers borrow some techniques from solving Boolean Satisfiability (SAT) problems. These SMT solvers incorporate an *off-the-shelf* SAT solver for handling logical structure and many dedicated solvers for handling background theories. The performance of this architecture, in practice, increases dramatically, especially when a formula contains some symbols from a theory whose complexity is extremely high.

Many tools use SMT solvers as their underlying engines. SMT solvers were first proposed for solving formulas generated by verification tools, and therefore many verification tools depend on them, e.g., KIND [40]. BOOGIE [9] uses an SMT solver, called Z3 [28]. Furthermore, there are many verification tools, e.g., DAFNY [54] and VCC [26], translating their original languages into the BOOGIE language. Other applications include: Interactive Theorem Proving (e.g., PVS [69]), Predicate Abstraction [48], Test Case Generation (e.g., Microsoft USB Test Tool [28]), and so on.

1.3 Contributions

Existing string solvers (see Chapter 5) are good at solving constraints over some particular fragments, e.g., fixed-length string constraints (i.e., every string variable has a fixed constant length). However, they are either unsound, e.g., KALUZA [82] and Z3-STR [98, 89] (see Section 3.4), or lack of expressiveness, e.g., REX [91] and HAMPI [51].

Our work devises an algebraic approach for solving constraints over a theory of unbounded strings with length and symbolic regular expressions natively. This

approach can be used to add capacity to DPLL(T)-based general, multi-theory SMT solvers to reason about string constraints. Indeed, we have integrated a standard DPLL(T) string theory solver to CVC4 [10].

More specifically, our contribution to automated reasoning over string constraints can be summarized into the following points:

Expressive Language. Constraints from security applications contain not only string constraints but also the ones from other theories, e.g., arithmetic and uninterpreted functions. Even if a constraint is from the theory of strings, it may contain membership and other string manipulating functions, e.g., extraction of substrings. To the best of our knowledge, existing string solvers can only handle a small fragment of such constraints. We have devised a new algorithm that solves quantifier-free constraints over a theory of unbounded strings with length, membership, and common string manipulating functions. Indeed, we introduce symbolic regular expressions in our language which allow us to reason about membership constraints over a non-regular language. In addition, our approach integrates a specialized string theory solver for such constraints within the DPLL(T) framework. It allows our string solver to accept an even richer language over extended theories. Furthermore, our tool is an underlying string constraint solver for many security vulnerability detection projects, e.g., the Artemis project [4].

Proved Soundness. A string solver is usually used as the underlying constraint

[4]Artemis is a tool that performs automated, feedback-directed testing of JavaScript applications, available at `https://github.com/cs-au-dk/Artemis`.

solver in security checking tools, and thus is essential for a security checking tool. Furthermore, a string solver usually provides a certificate, for its front-end tools, either to prove the functional correctness or to disprove the correctness by generating a counter-example. If a string solver is questionable, its related tools are doomed to be unreliable. With respect to our theoretical contribution, we provide detailed proofs for the soundness of the calculus we developed to formalize our approach. To be more specific, our soundness proofs include

- *refutation soundness*: given a set of constraints, if our calculus produces a *closed* derivation, the constraints are unsatisfiable;
- *solution soundness*: given a set of constraints, if our calculus produces a *saturated* derivation, the constraints are satisfiable and the induced model is indeed a model for the constraints.

Solution Completeness. The satisfiability of word equations is related to the Hilbert's tenth problem [62]. It was proved to be decidable by Makanin [60], and was later refined to be a PSPACE problem by Plandowski [72]. Combined with length constraints, the decidability is still an open question [36]. With the extension of string manipulating functions, e.g., replacement and substring, the problem has been shown to be undecidable [17]. Due to these theoretical difficulties, we have not proved the *refutation completeness* of our algorithm over the whole theory. Instead, our contribution to completeness is that we have proved the *solution completeness* of our calculus under a *fair* derivation strategy, i.e., if the problem is satisfiable, with

this strategy any derivation produced is guaranteed to be saturated.

Decision Procedure. A decision procedure is important for automated reasoning. It guarantees the termination of a procedure on any problem over a theory. Our contributions in this direction include the following:

- we have identified a non-trivial fragment in the theory of unbounded strings with length constraints, based on a certain *acyclic* condition on the input problem. We provide a description of this fragment, as well as the proof that shows our derivation is guaranteed to provide a correct answer on any problem in this fragment;
- we also provide a proof that our procedure is a decision procedure for membership and length constraints.

High Performance. The crucial part in finding a vulnerability in a program is to reason about string constraints from symbolic execution. The efficiency of solving string constraints determines the success of a vulnerability detection. Therefore, performance is an important assessment for an automated reasoning tool. Recent studies [46] show that most existing procedures are not efficient enough, when a problem becomes big. We have implemented a highly efficient string solver, and our initial results demonstrate that our approach outperforms the other string solvers in terms of correctness, precision, and run time.

1.4 Overview

This dissertation is organized as follows: In Chapter 2, we provide the technical background of this thesis. We introduce the terminologies for monoids, strings and languages, and discuss the DPLL procedure, the Nelson-Oppen combination, the DPLL(T) procedure and the existential theories over string constraints. In Chapter 3, we introduce the satisfiability problem of string constraints and our calculus for solving this problem. We also discuss the correctness, solution completeness and the experimental results with respect to the calculus. In addition to [56], we introduce our new refinements and the extension to string manipulating functions. In Chapter 4, we introduce our decision procedure for solving membership with length constraints. We also provide the soundness and completeness proof. In addition, we discuss an extension to solve the symbolic membership constraints. In Chapter 5, related work is discussed. Finally, a summary and some potential directions for future research in this topic are presented in Chapter 6.

CHAPTER 2

BACKGROUND AND PRELIMINARIES

In this chapter, we introduce several formal concepts of the theory of monoids, strings and formal languages. We also establish some general notation for later chapters.

We use $\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{S}$ to represent *Boolean values*, *Natural Numbers*, *Integers* and *Strings*, respectively. A *language* is a set of strings that can be constructed from a set of symbols by a grammar. Here, we use $\mathbb{L}$ to denote the class of *Languages* we consider throughout the dissertation. It can be partitioned into two sub-classes: the class of *Symbolic Regular Languages* $\mathbb{L}_{\mathbb{V}}$ and the class of *Concrete Regular Language* $\mathbb{L}_{\mathbb{C}}$, i.e., $\mathbb{L} = \mathbb{L}_{\mathbb{V}} \cup \mathbb{L}_{\mathbb{C}}$ and $\mathbb{L}_{\mathbb{V}} \cap \mathbb{L}_{\mathbb{C}} = \emptyset$. The difference between these two classes is whether a string variable is allowed in a language, i.e., $L \in \mathbb{L}_{\mathbb{C}}$ iff $\mathcal{V}(L) = \emptyset$ and $L \in \mathbb{L}_{\mathbb{V}}$ iff $\mathcal{V}(L) \neq \emptyset$, where the function $\mathcal{V}$ takes a language and returns a set containing all free variables in that language.

Organization. In Section 2.1, we review the properties of monoids. In Sections 2.2 and 2.3, we discuss the properties of strings and languages, respectively. In Section 2.4, we introduce the many-sorted first-order logic. In Section 2.5, we provide the background for the DPLL procedure, the Nelson-Oppen combination and the DPLL(T) procedure. In the last section, we discuss the background for existential theories over string constraints.

2.1 Monoids

In abstract algebra, a *semigroup*, denoted as $\langle S, o\rangle$, is a set of elements S and an associative binary operation o that is built upon this set. A semigroup is *closed* under its operation o, i.e., $o/2 : S \times S \to S$ [1]. The operator o is *associative*, i.e., $o(a, o(b, c)) = o(o(a, b), c)$, where $a, b, c \in S$.

The *cardinality* of a set S is the number of elements in S, denoted as $|S|$. In general, the cardinality can be infinite. In the context of the theory of strings, the set (of a monoid) is often referred to an alphabet, and we only consider an alphabet that is arbitrary large but finitely many. Thus, the cardinality of an alphabet is always a natural number. In the context of the theory of languages, we override the function symbol for the language sort. Since the cardinality of a language may be infinite, we use a constrained formula that represents the cardinality of the language.

Given two semigroups $G_1 = \langle S_1, o_1\rangle$ and $G_2 = \langle S_2, o_2\rangle$, a semigroup *morphism* is a *total* (unary) function $f/1 : G_1 \to G_2$. This function also preserves the semigroup structure, i.e., $f(o_1(a, b)) = o_2(f(a), f(b))$, where $a, b \in G_1$.

A semigroup $\langle S_1, o_1\rangle$ is a *sub-semigroup* of $\langle S_2, o_2\rangle$, if $S_1 \subseteq S_2$, $o_1 = o_2$, and the semigroup $\langle S_1, o_1\rangle$ is closed.

An *identity* element $\epsilon^{S,o}$ (with respect to a set S and an operator o) is a special element in S, such that this element is always absorbed by any other element under the operation o, i.e., $\forall a \in S.\ o(a, \epsilon^{S,o}) = o(\epsilon^{S,o}, a) = a$. Normally [75], this element is also called *two-sided identity* element. Throughout this dissertation, we always refer

[1] $o/2$ indicates that the arity of the function o is 2.

an identity as a two-sided identity element. Moreover, when the context is clear, we omit the superscript of an identity.

A *monoid* M is a semigroup $\langle S, o\rangle$ with an identity ϵ in S. Similar to semigroup, a *sub-monoid* of M is a monoid where the element set is a subset of the elements of M and it is closed under the same operation o. By this definition, a sub-monoid always keeps the same identity as the one in M.

Given two monoids M_1, M_2, a monoid *morphism* $f/1 : M_1 \to M_2$, is a total function that preserves the monoid structure. In addition, this function maps the identity in M_1 to the identity in M_2. A bijective monoid homomorphism is called *isomorphism*. Two monoids M_1 and M_2 are called isomorphic if there is a monoid isomorphism between them.

Example 2 The pair $\langle \mathbb{S}, \cdot\rangle$ is a monoid, where $\mathbb{S}$ is the set of strings, $\cdot$ is the concatenation operator, and the empty string is the identity. $\langle \mathbb{N}, +\rangle$ is a monoid, where $\mathbb{N}$ is the set of natural numbers, $+$ is the standard plus operator, and 0 is the identity. The length function $|\_|$, which maps a string to its length, is a monoid morphism. □

Given a set S and an associative operator o (S does not contain an identity with respect to o), a unique set U can be generated by the following two rules:

- $s \in U$, if $s \in S$;
- $o(s, t) \in U$, if $s, t \in U$.

The generated set together with the operator $\langle U, o\rangle$ is called a *free semigroup* on the set S, and it is denoted as S^+. $\langle U \cup \{\epsilon\}, o\rangle$ is called a *free monoid* on the set S, and

it is denoted as S^*. The set S is called the *generating set* of U.

Proposition 1 (Universal Property) Given a homomorphism ϕ that maps a set S to a monoid M, there exists a unique monoid morphism ψ that maps S^* to M. □

The proof is given in [27]. As a consequence, we have the following alternative definition of free monoids. Notice that not all monoids are free.

Corollary 1 A monoid M is free if it has a generating set S and an isomorphism between S^* and M. □

The *product* of the two sets S_1 and S_2 with respect to an operator o, is defined as a set: $S_1 \cdot_o S_2 = \{o(s,t) \mid s \in S_1, t \in S_2\}$. If the context is clear, we omit the subscript of the product.

Proposition 2 (Minimal Generating Set) Given a monoid $M = \langle S, o \rangle$, a unique set T can be generated by $T = S\backslash\{\epsilon\}\backslash((S\backslash\{\epsilon\}) \cdot_o (S\backslash\{\epsilon\}))$ [2]. This set T is called the *minimal generating set* of M. □

It is obvious that the set T can generate the monoid M and any proper subset set of T cannot be a generating set of M.

2.2 Strings

An *alphabet* $\mathcal{A}$ is any non-empty finite set. Each element in $\mathcal{A}$ is called a *character*. Unless explicitly stated, we do not distinguish a character from a string. We refer a character to be a string of length one. We use $\mathbb{S}_\mathcal{A}$ or $\mathcal{A}^*$ to denote the

[2] The operator $\backslash$ is for the set difference by standard semantics.

set of all string with respect to the alphabet $\mathcal{A}$. For simplicity, if the context (of the alphabet) is clear, we use $\mathbb{S}$ for the set.

A *string* s over $\mathcal{A}$ is a finite sequence of characters over $\mathcal{A}$, i.e.,

$$s^{\mathcal{A}} = c_0^{\mathcal{A}} c_1^{\mathcal{A}} \cdots c_{n-1}^{\mathcal{A}}.$$

We use ϵ to denote the *empty string*, where the length of the sequence is zero. Since we only consider string operations over the same alphabet in this dissertation, we omit the alphabet symbol for string terms. We use $s[i]$ to denote the character at the position i in s, e.g., $s[1] = c_1$ if $s = c_0 c_1 \cdots c_{n-1}$[3].

The operation *concatenation*, $\cdot/2 : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$, takes two strings s, t, and returns a new string which starts with the sequence of characters in s, followed by the sequence of characters in t, i.e., $s \cdot t = c_0 c_1 \cdots c_{n-1} d_0 d_1 \cdots d_{n-1}$, where $s = c_0 c_1 \cdots c_{n-1}$ and $t = d_0 d_1 \cdots d_{n-1}$ [4]. String concatenation is an associative operation, but non-commutative. The empty string is the identity over the concatenation operation.

Strings together with the concatenation operation is a monoid, which satisfies the following three properties:

- *Closure*: $\forall s, t \in \mathbb{S}.\ s \cdot t \in \mathbb{S}$,
- *Associativity*: $\forall s, t, r \in \mathbb{S}.\ (s \cdot t) \cdot r = s \cdot (t \cdot r)$, and
- *Identity*: $\forall s \in \mathbb{S}.\ \epsilon \cdot s = s \cdot \epsilon = s$.

[3]Index starts from 0.

[4]Technically speaking, it is not formal enough and an inductive proof is needed. Since this proof is well-known, we omit the proof for simplicity.

Because of the associativity, we can always write $s_1 \cdot s_2 \cdot s_3$ for either $s_1 \cdot (s_2 \cdot s_3)$ or $(s_1 \cdot s_2) \cdot s_3$.

A string s is always associated with a natural number, a.k.a. the *length*, which indicates the number of characters in s, i.e., $|s| = n$ iff $s = c_0 c_1 \cdots c_{n-1}$. The length function $(|\_|/1 : \mathbb{S} \to \mathbb{N})$ is a monoid morphism between $\langle \mathbb{S}, \cdot \rangle$ and $\langle \mathbb{N}, + \rangle$:

- $|\epsilon| = 0$, and
- $|s \cdot t| = |s| + |t|$, for any strings s and t,

where $+$ is the standard addition operation over the natural numbers.

Two strings are *identical*, iff they have the same length and all characters (at same positions) are identical, i.e., $s = t$ iff $|s| = |t|$ and $\forall 0 \leq i < |s|.\ s[i] = t[i]$.

The symbol $\mathcal{A}^n$ denotes all strings over the alphabet $\mathcal{A}$ of length n, e.g., $\mathcal{A}^0 = \{\epsilon\}$, $\mathcal{A}^1 = \mathcal{A}$, and $\mathcal{A}^2 = \{aa, ab, ba, bb\}$, assuming $\mathcal{A} = \{a, b\}$. Therefore, the set of all strings, denoted as $\mathcal{A}^*$ or $\mathbb{S}$, is $\bigcup_{i \in \mathbb{N}} \mathcal{A}^i$, and the set of all non-empty strings, denoted as $\mathcal{A}^+$, is $\bigcup_{i \in \mathbb{N} \setminus \{0\}} \mathcal{A}^i$.

Similarly, s^n denotes the concatenation of the string s for n times, e.g., $s^3 = s \cdot s \cdot s$. When $n = 0$, s^0 is the empty string.

Given a free sub-monoid $\langle S, \cdot \rangle$ and its minimal generating set T, if any string in S has a unique representation over T (i.e., if $a_0 a_1 \cdots a_{n-1} = b_0 b_1 \cdots b_{m-1}$ and $a_i, b_j \in T$, then $m = n$ and $\forall i.\ a_i = b_i$), the set T is called a *code* of S.

A string s is a *substring* of t if there exist two strings u and v, such that $t = u \cdot s \cdot v$. If $s \neq t$, we call s is a *proper substring* of t. A string s is a *prefix* of t if

there exists a string v, such that $t = s \cdot v$. If $s \neq t$, we call s is a *proper prefix* of t. A string s is a *suffix* of t if there exists a string u, such that $t = u \cdot s$. If $s \neq t$, we call s is a *proper suffix* of t.

Proposition 3 (Fine and Wilf Theorem) Given two strings u and v, d is the greatest common divisor of $|u|$ and $|v|$, if u^p and v^q have a common prefix of length of at least $|u| + |v| - d$, u and v are the power of the same string of length d. □

This theorem is proved in [34], and a more general extension is given in [61].

The *reverse* of string $c_0c_1 \cdots c_{n-2}c_{n-1}$ is the string $c_{n-1}c_{n-2} \cdots c_1c_0$. A string s is called a *palindrome* if the string s is identical to its reverse.

Proposition 4 (Levi's Lemma) Given four strings u, s, v, t, if $u \cdot s = v \cdot t$, then there exists another string k, such that either $u = v \cdot k$ or $v = u \cdot k$. □

This lemma is also called the *equidivisibility* theorem. It provides a basis for some rules of our calculus in later chapters while we are solving word equations.

A non-empty string s is *primitive* if it is not a power of any of another string, i.e., $s = t \vee s \notin \{t^n \mid n \geq 0\},\ \forall t \in \mathbb{S}$.

Proposition 5 Given two strings s and t, if $s^p = t^q, p > 0, q > 0$, there exists a unique primitive string k, such that $s, t \in \{k^n \mid n \geq 0\}$. □

Two strings s and t are *conjugate*, if there exist $u, v \in \mathbb{S}$, such that $s = u \cdot v$ and $t = v \cdot u$.

Proposition 6 If two strings s and t are conjugate, there exists a string $k \in \mathbb{S}$, such that $s \cdot k = k \cdot t$. □

Given an ordering $\prec$ on an alphabet, the lexicographic order $\prec_{lex}$ on strings is defined as follows: a string s is less then a string t in the lexicographic order iff

- s is a proper prefix of t, or
- $s = u \cdot a \cdot v$ and $t = u \cdot b \cdot v'$, where a and b are two characters, such that $a \prec b$.

2.3 Languages

A *formal language* is a set of strings that can be expressed by a set of rules. In this dissertation, we focus on the class of *concrete regular languages* and its extension, the class of *symbolic regular languages*.

A *concrete regular language* is a set of strings denoted by a regular expression R, i.e., $\mathcal{L}(R)$. A *regular expression* consists of a set of string singletons and a set of set operators.

In the base cases, given an alphabet $\mathcal{A}$, the following sets can be a regular expression:

- $\emptyset_R$: an *empty set* that does not accept any string, i.e., $\mathcal{L}(\emptyset_R) = \emptyset$;
- s_R : a *singleton set* that contains only s, i.e., $\mathcal{L}(s_R) = \{s\}$, where $s \in \mathbb{S}$.

If R_1 and R_2 are regular expressions, the following expressions are also regular expressions:

- *Concatenation* : $R_1 \cdot R_2 = \{s \cdot t \mid s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\}$,

- *Alternation* : $R_1 \cup R_2 = \{s \mid s \in \mathcal{L}(R_1) \cup \mathcal{L}(R_2)\}$,
- *Intersection* : $R_1 \cap R_2 = \{s \mid s \in \mathcal{L}(R_1) \cap \mathcal{L}(R_2)\}$, and
- *Kleene Star* : $R_1^* = \bigcup_{i \in \mathbb{N}} \mathcal{L}(R_1^i)$.

Furthermore, if R_1 and R_2 are regular expressions and $\preceq_{lex}$ is a total ordering on the alphabet $\mathcal{A}$, the following additional expressions are also regular expressions:

- *Option* : $R_1^? = \{\epsilon\} \cup \mathcal{L}(R_1)$,
- *Range* : $[a - b] = \{c \mid a \preceq_{lex} c \preceq_{lex} b, c \in \mathbb{S}\}$, where a, b are two characters [5],
- *Loop* : $R_1^{n,m} = \bigcup_{n \leq i \leq m} \mathcal{L}(R_1^i)$,
- *Plus* : $R_1^+ = \bigcup_{i \in \mathbb{Z}^+} \mathcal{L}(R_1^i)$, and
- *Complement* : $R_1^c = \{s \mid s \notin \mathcal{L}(R_1)\}$.

A regular expression is recognized by a finite automaton, if every string in the language of the regular expression is accepted by the finite automaton.

A *finite automaton* (FA) can be described as a tuple $\langle Q, \mathcal{A}, \delta, q_0, F \rangle$, where

- Q is a finite set of states,
- $\mathcal{A}$ is the alphabet,
- δ is called the transition function,
- q_0 : is a start state, and $q_0 \in Q$, and

[5] A character is a string of length 1.

- F : is a set of accept states, and $F \subseteq Q$.

If the transition function δ is a total function that maps a state and a character to another state, i.e., $\delta : Q \times \mathcal{A} \to Q$, the finite automaton is called the *Deterministic Finite Automaton* (DFA). If the transition function δ maps to a power-set of states, i.e., $\delta : Q \times \mathcal{A} \to \mathcal{P}(Q)$, the finite automaton is called the *Nondeterministic Finite Automaton* (NFA).

A *concrete regular language* is a set of strings recognized by a concrete regular expression. A *symbolic regular language* is a set of strings recognized by a symbolic regular expression R. The difference between a concrete regular expression and a symbolic regular expression is that the symbolic one allows string variables in its singleton set. A string is in a symbolic regular language if it is in one instance of the symbolic regular language. However, by allowing string variables in a language, the language accepted by a symbolic regular expression is not regular.

Example 3 The language of the symbolic regular expression $(x \cdot a \cdot x)^*$ is not context-free, where a is a character, x is a string variable, and the cardinality of alphabet is greater than 1. It can be shown by the pumping lemma. □

Proposition 7 (Equivalent Expressiveness) DFA and NFA have the equivalent expressiveness, i.e., any language that is recognized by a NFA can also be recognized by a DFA, and vise versa. □

Although the expressiveness of DFA and NFA is the same by Proposition 7, it is important to notice the following facts:

- it is easier to construct an NFA, and to recognize a same language, an NFA is usually smaller than a DFA;
- it is more efficient to execute a DFA;
- conversion from an NFA to a DFA causes an exponential growth of state size.

Proposition 8 (Kleene's Theorem) A language is regular iff there exists a finite automaton that recognizes it. □

A regular language is defined by its corresponding regular expression. As a corollary to the Kleene's theorem, we can construct a regular expression from a FA, and vice versa.

Proposition 9 (Closure Properties) The class of concrete regular languages are closed under Union, Intersection, Difference, Concatenation, Kleene Closure, Complementation, Reversal, Homomorphism, Inverse Homomorphism operations. □

A reader can find that the closure properties are proved by automaton construction in [47].

The *derivative* of a regular expression R with respect to a character c returns a language that contains all suffixes where the prefix of those original strings is c, i.e., $\partial_c R = \{u \mid c \cdot u \in \mathcal{L}(R)\}$.

Proposition 10 The language returned by the derivative is still regular. □

This derivative function can be extended to a string as its argument, instead of a single character, i.e., $\partial_s R = \{u \mid s \cdot u \in \mathcal{L}(R)\}$.

Corollary 2 A string s is accepted by a regular expression R iff the empty string is in the language of $\partial_s R$. □

2.4 Many-Sorted First-order Logic

We work in the context of *many-sorted first-order* logic with *equality* [12]. A many-sorted *signature* Σ contains a set of sort symbols and a set of sorted function symbols. A sorted function symbol is $f/n : S_0 \times S_1 \times \cdots S_{n-1} \to S_n$, where S_is are sorts in Σ and n is the arity of the function symbol f. A constant symbol is a special function symbol where the arity is 0. We use f instead of $f()$ to represent a constant.

Given a signature Σ, a Σ-term is either a variable or an expression in a form of $f(t_0, t_1, \cdots, t_{n-1})$, where f is an n-ary function symbol in Σ and t_i's are Σ-terms. A term is *well-sorted* with respect to a sort S if it is :

- a variable of the sort S, or
- a function term: $f(t_0, t_1, \cdots, t_{n-1})$, where $f/n : S_0 \times S_1 \times \cdots S_{n-1} \to S$ for $\forall 0 \leq i \leq n-1$. And t_i is a well-sorted term with respect to the sort S_i.

A Σ-*equality* is an equality $t_0 \approx t_1$ where both t_0 and t_1 are well-formed terms with respect to the same sort. A Σ-*atom* is either a Σ-equality, or a well-formed term with respect to the sort Bool. A Σ-*literal* is either a Σ-atom or its negation. A Σ-*clause* is a disjunction of Σ-literals. A Σ-*formula* is an expression that is built from a set of Σ-literals and logical connectives, such as $\wedge, \vee, \neg$.

A *ground* term is a Σ-term without any variables. This terminology is also extended to literals, clauses and formulas. A variable in a formula is a *free* variable

if it is not bounded by a quantifier; otherwise, we call it a *quantified* variable. We use $\mathcal{V}(\phi)$ to denote all free variables in the formula ϕ. If all variables in a formula are *free*, we call this formula a *quantifier-free* formula. Unless it is explicitly stated, a Σ-formula in this dissertation refers to a *quantifier-free* formula.

A Σ-*interpretation* $\mathcal{M}$ is a non-empty domain (denoted as M) and a mapping that associates:

- each variable x with an element $x^{\mathcal{M}}$ of M,
- each function symbol $f/n \in \Sigma$ with a function $f^{\mathcal{M}}/n : \mathsf{M}^n \to \mathsf{M}$, and
- each predicate symbol $p/n \in \Sigma$ with a relation $p^{\mathcal{M}} \subseteq \mathsf{M}^n$.

Given a Σ-interpretation $\mathcal{M}$ and a Σ-term t, the *semantics* of the term t, denoted as $\mathcal{M}[\![t]\!]$, is defined recursively as follows:

- if t is a variable x, $\mathcal{M}[\![x]\!] := x^{\mathcal{M}}$ (the assignment for x in $\mathcal{M}$)
- if t is in a form of $f(t_0, t_1, \cdots, t_{n-1})$,

$$\mathcal{M}[\![f(t_0, t_1, \cdots, t_{n-1})]\!] := f^{\mathcal{M}}(\mathcal{M}[\![t_0]\!], \mathcal{M}[\![t_1]\!], \cdots, \mathcal{M}[\![t_{n-1}]\!]).$$

Given a Σ-interpretation $\mathcal{M}$ and a Σ-formula ϕ, the formula ϕ is *satisfiable with respect to* $\mathcal{M}$, denoted as $\mathcal{M} \models \phi$, if it is one of the followings:

- $\mathcal{M} \models p(t_0, \cdots, t_{n-1})$ iff $\langle \mathcal{M}[\![t_0]\!], \cdots, \mathcal{M}[\![t_{n-1}]\!] \rangle \in p^{\mathcal{M}}$
- $\mathcal{M} \models t_1 \approx t_2$ iff $\mathcal{M}[\![t_1]\!] = \mathcal{M}[\![t_2]\!]$
- $\mathcal{M} \models \phi \wedge \psi$ iff $\mathcal{M} \models \phi$ and $\mathcal{M} \models \psi$

- $\mathcal{M} \models \phi \vee \psi$ iff $\mathcal{M} \models \phi$ or $\mathcal{M} \models \psi$
- $\mathcal{M} \models \neg\phi$ iff $\mathcal{M} \not\models \phi$

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations. Σ-interpretations are also called the *models* of T. The class of Σ-interpretations $\mathbf{I}$, is closed under variable reassignment.

A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas *entails in* T a Σ-formula φ, written as $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is *satisfiable in* T if $\Gamma \not\models_T \bot$ where $\bot$ is the universally false atom. We will write $\Gamma \models \varphi$ to denote that Γ entails φ in the class of all Σ-interpretations.

We write $s \not\approx t$ as an abbreviation of $\neg\, s \approx t$. If e is a term or a formula, we denote by $\mathcal{V}(e)$ the set of e's free variables, extending the notation to tuples and sets of terms/formulas as expected.

Two Σ-formulas are *equisatisfiable in* T if for every model $\mathbf{I}$ of T that satisfies one formula, there is a model of T that satisfies the other one, and the differences of those two models from $\mathbf{I}$ are, at most, over the free variables that are not shared by the two formulas.

2.5 DPLL(T) Procedure

Modern SMT solvers use the DPLL(T) [66] architecture that delegates decisions about the satisfiability of formulas in a specific theory to dedicated theory solvers. State-of-the-art SMT solvers include Z3 [28] and CVC4 [10]

2.5.1 The DPLL Procedure

In propositional logic, a formula is constructed from a set of Boolean variables using a set of logical connectives, such as $\wedge, \vee, \neg$. Boolean variables can be assigned a (truth) value that is either true or false. Given a Boolean formula, the Boolean satisfiability (SAT) problem is to answer whether we can find an assignment for those variables, such that the formula is evaluated to be true under the assignment.

The SAT problem is a classical NP-complete problem. There are many decision procedures to solve this problem (e.g., Binary Decision Diagram [29]). Among them, the DPLL [6] procedure [65] is the most frequently used in most modern SAT solvers.

The DPLL procedure takes a set of clauses [7] as input. It returns sat if a logically consistent assignment [8] can be found; otherwise, it returns unsat. During its computation, the procedure maintains an internal stack of literals (possibly with decision marks) to represent a partial assignment. The stack contains two sorts of elements:

- *Decision literal*: a variable assignment introduced by guess [9],
- *Propagation literal*: a variable assignment introduced by logical deduction.

Whenever every clause is evaluated to be true under the current assignment, the pro-

[6] DPLL is named after its authors: Davis, Putman, Logemann and Loveland.

[7] A clause is a disjunction of Boolean variables.

[8] An assignment is consistent with the clause set if all clauses are evaluated to be true under this assignment.

[9] If a decision literal is the nearest to the bottom, we call it the *first decision literal*.

cedure stops and returns sat. During a standard processing loop, the DPLL procedure processes the clause set in three steps:

1. It first checks whether the current partial assignment is consistent with the clause set by evaluation. If one of the clauses is evaluated to false and the stack contains at least one decision literal, the procedure pops out all literals in the stack till the nearest decision literal, then flips the sign of that decision literal and turns it into a propagation literal. This step is often referred to Backtrack. If one of the clauses is evaluated to false and the stack does not contain one decision literal, the procedure stops and returns unsat.

2. After the inconsistent check, the procedure tries to push new propagation literals into the stack by logical deduction, e.g., Unit Propagation [97].

3. If there are still unassigned variables, the procedure picks one heuristically, guesses its sign [10], and pushes it into the stack as a decision literal.

The procedure continues until all variables are assigned.

The DPLL procedure can be further extended to the Conflict-Driven Clause Learning (CDCL) procedure by

- replacing the naive backtracking with backjumping [11],
- introducing lemma learning, and

[10] the polarity of a literal

[11] a more efficient algorithm that allows the procedure backtracks more than one level when a conflict occurs

- adding modern engineering techniques, e.g., two watched literals.

2.5.2 The Nelson-Oppen Combination

In the above paragraphs, we discussed the satisfiability of Boolean logic formulas. In Boolean formulas, the signature only contains propositional variables and logical connectives; whereas in SMT formulas, the signature is extended to a set of predicate symbols, a set of function symbols and a set of non-Boolean variables. Those extended symbols can be interpreted in some background theories. An SMT formula is satisfiable if there exists a model that satisfies both the logical formula and the background theories.

Reasoning about an SMT formula usually involves several reasoning over several theories. We refer the procedure to reason over a theory T as a T-solver. Note that a T-solver can only handle conjunctions of literals. Given a formula over several theories where each theory has a T-solver, the Nelson-Oppen combination provides a procedure to reason about this constraint.

Before discussing the Nelson-Oppen theory, we introduce two more definitions. A Σ-theory T is *stably infinite* iff every T-satisfiable quantifier-free Σ-formula has a T-interpretation whose domain is infinite.

Given a sub-signature $\Sigma_0 \subseteq \Sigma$, a Σ-theory T is *convex* iff whenever $T \models \bigwedge_{i=1}^{n} a_i \implies \bigvee_{j=1}^{m} b_j$, there exists $k \in \{1, \cdots, m\}$ such that $T \models \bigwedge_{i=1}^{n} a_i \implies b_k$, where a_i's are Σ-atoms and b_j's are Σ_0-atoms.

Proposition 11 (Deterministic Nelson-Oppen) Given a set of stably-infinite and convex theories, the satisfiability of a constraint over these theories can be checked

by the deterministic Nelson-Oppen combination. □

Without loss the generality, we describe the deterministic Nelson-Oppen combination with two stably-infinite and convex theories T_1 and T_2. We use S_1 and S_2 to denote the literals for T_1-solver and T_2-solver, respectively. Initially, constraints are partitioned into S_1 and S_2 based on their signature. Whenever S_1 is T_1-unsatisfiable or S_2 is T_2-unsatisfiable, the original problem is unsatisfiable. If T_1-solver derives a new T_2-equality from S_1 but not in S_2 yet, the new constraint will be put in S_2; similarly, if T_2-solver derives a new T_1-equality from S_2 but not in S_1 yet, the new constraint will be put in S_1. If no new equality can be derived from either T-solver and both S_1 and S_2 are T-satisfiable, the original problem is satisfiable.

Note that not every theory is *convex*. For example, integer linear arithmetics is not convex with respect to inequality. If a theory combines with a non-convex theory, the deterministic Nelson-Oppen combination will not work. However, the non-deterministic Nelson-Oppen algorithm is for non-convex theory combination.

Let C be a set of constants. An *arrangement* A over C is a set of (dis-)equalities, such that for every pair of elements in C, either an equality or a disequality (between two) is in A.

Let T_1 and T_2 be two stably-infinite theories, the non-deterministic Nelson-Oppen algorithm work as follows: initially, constraints are partitioned into S_1 and S_2 based on their signature; for every arrangement A over the shared constants, if both $S_1 \cup A$ and $S_2 \cup A$ is T-satisfiable, the original problem is satisfiable; otherwise, it is unsatisfiable.

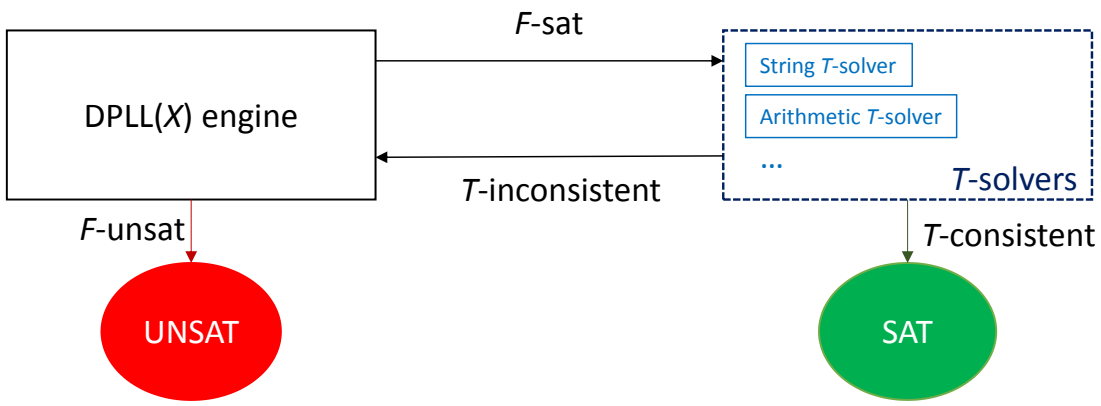

Figure 2.1. A general DPLL(T) architecture.

Therefore, the Nelson-Oppen combination procedure [63] is a decision procedure for combined theories, when those theories are stably infinite. Note that when a combined theory consists of a non-stably infinite theory, a generalized Nelson-Oppen combination method is provided in [88]. With respect to the word problems, the stably infiniteness is discussed in [7].

2.5.3 The DPLL(T) Architecture

The DPLL(T) approach is a lazy approach to solve the satisfiability of a SMT formula. It uses the Nelson-Oppen theory as its basis. The DPLL(T) architecture can be divided into two parts, as shown in Figure 2.1 : the logic solving part and the theory solving part.

In the logic solving part, it usually refers to the DPLL(X) procedure. A DPLL(X) engine is a modified CDCL-based SAT solver. Similar to a CDCL-based solver, a DPLL(X) engine maintains a partial model M and the current formula

F [12]. The major difference between a DPLL(X) engine and a generic DPLL-based SAT solver is that a DPLL(X) engine

- requires the mechanism for incremental clause assertions [13]), and
- disables some engineering optimizations (e.g., pure literals [65]).

The theory solving part usually contains several dedicated solvers for background theories, called T-solvers. Compared to a generic theory solver, a T-solver requires the mechanisms for incrementality and backtracking. A T-solver maintains a set of T-related literals. This set is a subset of the partial assignment M.

Initially, the DPLL(X) engine gets the formula F from the input. It tries to find a model for the literals. If failed, it returns unsat; otherwise, it distributes each literal in M to a corresponding T-solver according to some T-signature-based heuristics.

When a T-solver gets a set of literals, it first checks whether these literals are consistent with the theory. If it is T-consistent, the T-solver does nothing but reports *consistent* back to the main engine. If it is T-inconsistent, it either returns a conflict (in a form of literal conjunction) [14], or propagates a new literal with an explanation (in a form of implication) [15].

---

[12] F may be changed during computation.

[13] Most CDCL-based SAT solvers have the incremental mechanism because of clause learning (e.g., MINISAT.

[14] It is called T-unsatisfiable.

[15] It is called T-entail.

If all T-solvers report T-consistent, the procedure stops and answers sat with the model M; otherwise, the DPLL(X) engine collects all clauses returned by T-solvers, and asserts them into the formula F. Then, it tries to build a model from this new formula.

2.6 Existential Theories over Strings

A first-order formula is in *prenex* form if all quantifiers appear in the front of a *quantifier-free* formula, i.e.,

$$\forall \overline{X_1}.\ \exists \overline{X_2}.\ \cdots .\ \phi[\overline{X_1}, \overline{X_2}, \cdots], \tag{2.1}$$

where ϕ is a *quantifier-free* formula. Any (first-order) formula can be converted to a *prenex* form in linear time. Without loss of generality, we always refer a formula to a prenex form. If a formula (in the *prenex* form) contains only existential quantifiers, we call it an existentially quantified formula, or a constraint over an existential theory. If an existentially quantified formula contains no free variable, the formula is *valid* in T iff there exists a model M in T and the corresponding quantifier-free formula is evaluated to be true in M. For this reason, we do not distinguish satisfiability problems with existential theories.

We use String, Lan and Int to denote the String sort, the Language sort and the Integer sort, respectively. We use Σ_{SRL} to denote the signature over string constraints, where the subscript S is for symbols over the (pure) String sort, R for symbols over the Language sort, and L for the Length symbol. We use T_{X} to denote the existential theory over the signature Σ_{X}, where X is any combination of S, R and L. In particular, T_{S} refers to the existential theory over *word equations*, T_{SL} refers to the existential

theory over word equations and additional length constraints, T_{RL} refers to the existential theory over membership constraints and additional length constraints, and T_{SRL} refers to the full existential theory over strings. Throughout the whole thesis, unless explicitly stated, we focus on solving *quantifier-free* constraints.

The interpretations of T_{X} differ only on the variables. They all interpret Int as the set of integer numbers $\mathbb{Z}$, String as the set of all strings $\mathbb{S}$ over some fixed finite alphabet $\mathcal{A}$ of *characters*, and Lan as the power set of $\mathcal{A}^*$.

The common symbols (e.g., $+, -, \leq$) of linear integer arithmetic are interpreted as usual. The signature of the sort String (over word equations with length constraints) consists the following symbols:

- a constant symbol, or *string literal*, interpreted as the corresponding string in $\mathbb{S}$;
- a variadic function symbol $\mathsf{con} : \mathsf{String} \times \ldots \times \mathsf{String} \to \mathsf{String}$, interpreted as the string concatenation function $\cdot : \mathbb{S} \times \ldots \times \mathbb{S} \to \mathbb{S}$;
- a function symbol $\mathsf{len} : \mathsf{String} \to \mathsf{Int}$, interpreted as the string length function $|\_| : \mathbb{S} \to \mathbb{N}$.

The signature of the sort Lan consists the following symbols:

- a function symbol $\mathsf{set} : \mathsf{String} \to \mathsf{Lan}$, is interpreted as the function mapping a string $s \in \mathbb{S}$ to the language $\{s\}$;
- a constant symbol $\mathsf{rempty} : \mathsf{Lan}$, interpreted as the empty set $\emptyset$;

- a constant symbol $\mathsf{allchars} : \mathsf{Lan}$, interpreted as the set Σ;
- a variadic function symbol $\mathsf{rcon} : \mathsf{Lan} \times \ldots \times \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the language concatenation function $\cdot : \mathbb{L} \times \ldots \times \mathbb{L} \to \mathbb{L}$;
- a variadic function symbol $\mathsf{inter} : \mathsf{Lan} \times \ldots \times \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the set intersection function $\cap : \mathbb{L} \times \ldots \times \mathbb{L} \to \mathbb{L}$;
- a variadic function symbol $\mathsf{union} : \mathsf{Lan} \times \ldots \times \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the set union function $\cup : \mathbb{L} \times \ldots \times \mathbb{L} \to \mathbb{L}$;
- a function symbol $\mathsf{star} : \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the Kleene closure operator $\_^* : \mathbb{L} \to \mathbb{L}$;
- a predicate symbol $\mathsf{in} : \mathsf{String} \times \mathsf{Lan}$, interpreted as the set membership predicate $\in : \mathbb{S} \times \mathbb{L} \to \mathbb{B}$;
- a function symbol $\mathsf{opt} : \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the language option operator $\_^? : \mathbb{L} \to \mathbb{L}$;
- a function symbol $\mathsf{range} : \mathsf{String} \times \mathsf{String} \to \mathsf{Lan}$, interpreted as the language range operator $[\_,\ \_] : \mathbb{S} \times \mathbb{S} \to \mathbb{L}$;
- a function symbol $\mathsf{loop} : \mathsf{Lan} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Lan}$, interpreted as the language loop operator $\_^{\_,\ \_} : \mathbb{L} \times \mathbb{N} \times \mathbb{N} \to \mathbb{L}$;
- a function symbol $\mathsf{plus} : \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the language plus operator $\_^+ : \mathbb{L} \to \mathbb{L}$;

- a function symbol $\mathsf{comp} : \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the language complement operator $\_^c : \mathbb{L} \to \mathbb{L}$.

We call: *string term* any term of sort String or of the form $\mathsf{len}(s)$; *arithmetic term* any term of sort Int all of whose occurrences of len are applied to a variable; *regular expression* any term of sort Lan (possibly with variables). A *string constraint* is, a constraint containing at least one term of sort String. An *arithmetic constraint* is a (dis)equality $(\neg)s \approx t$ or an inequality $s > t$ where s and t are arithmetic terms. Note that if x and y are string variables, $\mathsf{len}\, x$ is both a string and an arithmetic term and $(\neg)\mathsf{len}\, x \approx \mathsf{len}\, y$ is both a string and an arithmetic constraint. A *membership constraint* is a literal of the form $(\neg)\mathsf{in}(s, r)$ where s is a string term and r is a regular expression. A SRL-*constraint* is a string, arithmetic or membership constraint. We will denote entailment in T_{SRL} simply as $\models_{\mathsf{SRL}}$.

Let $\mathcal{A}$ be a *finite* set of characters (a.k.a. alphabet). An *atomic string term* is either a *string literal* or a *string variable*. We say a string term over the alphabet $\mathcal{A}$ is *flattened*, if it is either an atomic string term or a *non-atomic flattened string term*. A *non-atomic flattened string term* is a concatenation of at least two non-empty atomic string terms, i.e.,

$$\mathsf{con}(u_0, u_1, \cdots, u_{n-1}), \tag{2.2}$$

where $n > 1$ and u_i's are either a non-empty string literal or a string variable, $\forall 0 \leq i < n$. A flattened string does not contain nested concatenations.

When a string contains nested concatenations, we call it a *non-flattened* string. An arbitrary *non-flattened* string term can be converted to a *flattened* string term via

rewrite rules, as given in Figure 3.1 [16]. For simplicity, when a concatenation contains single atomic string, we remove the top-level concatenation symbol, i.e., $\mathsf{con}(a) \rightarrow a$, where a is an atomic string term.

A *word equality* is an equality between two string terms. Since any string term can be flattened due to the associativity of concatenation, a word equality can be rewritten to an equality between two flattened string terms, i.e.,

$$\mathsf{con}(u_0, u_1, \cdots, u_{n-1}) \approx \mathsf{con}(v_0, v_1, \cdots, v_{m-1}), \tag{2.3}$$

where u_i's and v_j's are either a string constant or a string variable , $\forall 0 \leq i < n$, $\forall 0 \leq j < m$ [17]. A *word equation* is either a word equality or its negation (disequality). We use $\phi \dot{\approx} \psi$ to denote a word equation, where $\dot{\approx}$ is either an equality or a disequality. We use T_{S} to denote the existential theory of word equations.

A string *substitution* is a mapping from variables to terms of sort String. An *assignment* is a substitution that maps variables to string constants. Given a word equation $\phi \dot{\approx} \psi$, we call an assignment a *satisfying assignment* of the equation, if the assignment σ maps each variable (in both ϕ and ψ) to some string constant, and both sides are equal in T_{S}, i.e.,

$$\sigma \models_{T_{\mathsf{S}}} \phi \dot{\approx} \psi \quad \text{iff} \quad T_{\mathsf{S}} \models \sigma(\phi) \dot{\approx} \sigma(\psi), \tag{2.4}$$

where the symbol T_{S} refers to the theory of word equations.

---

[16] Further discussion is in Section 3.1.

[17] Note that an atomic string term u can also be considered as $\mathsf{con}(u)$ (if u is non-empty), or as $\mathsf{con}(\ )$ (if u is the empty string).

Given a set of word equations, the satisfiability problem (in T_{S}) is to determine whether there exists a model for this set of word equations: a model evaluates every equation to be true.

The satisfiability problem in string constraints has been attracting interest from both mathematicians and computer scientists for over a century [64]. There have been done extensive theoretic works on solving constraints over the theory of unbounded strings with length and membership constraints. In [36], an overview of decidable fragments is given. Detailed algorithms for word combinatorics are presented in [57, 58].

One major breakthrough was made by Markov, where he reduced the satisfiability problem of word equations to the Hilbert's tenth problem (the satisfiability of a Diophantine problem) [2]. However, the Diophantine problem was proved to be unsolvable by Matiyasevich [62].

A few years later, Makanin proved that a set of word equations can be reduced to a word equality and the satisfiability of word equations is decidable [60]. However, Makanin's algorithm, is highly impractical due to the equation size explosion at the very first step, where the procedure reduces a word disequality to a set of word equalities even although this procedure is still widely used in the latest work (e.g., [72, 73, 1]).

When a disequality appears, say $s \not\approx t$, the Makanin's algorithm converts the disequality to a disjunction of conjunctions of word equalities to represent all

possibilities, i.e.,

$$s \not\approx t \iff \bigvee_{a \in \mathcal{A}} s \approx \mathsf{con}(t, a, u) \vee t \approx \mathsf{con}(s, a, u) \vee \bigvee_{a,b \in \mathcal{A}, a \neq b} s \approx \mathsf{con}(w, a, u) \wedge t \approx \mathsf{con}(w, b, v), \tag{2.5}$$

where w, u, v are fresh variables and $\mathcal{A}$ is the alphabet.

Example 4 Assume the alphabet is $\{a, b, c\}$. Every disequality in the constraint set will introduce 3 fresh variables and a formula with 12 equalities. □

Note that if a constraint comes from a real application, the alphabet usually refers to the ASCII set where the cardinality is 256, not to mention Unicode. Introducing thousands of constraints at the first step is not an ideal choice.

Proposition 12 Let $S = \{s_1 \approx t_1, \cdots, s_n \approx t_n\}$ be a set of equalities, a and b be two distinct characters, and E be a single equality

$$\mathsf{con}(s_1, a, \cdots, s_n, a, s_1, b, \cdots, s_n, b) \approx \mathsf{con}(t_1, a, \cdots, t_n, a, t_1, b, \cdots, t_n, b). \tag{2.6}$$

S is satisfiable iff E is satisfiable. □

Although the proposition is used to reduce a set of equalities to a single one, it is more useful to apply in the opposite way, i.e., to split an equality to a set of equalities. In addition, disjunctions can be removed from a system by the algorithm in [49].

Proposition 13 Let S be a disjunction $s \approx t_1 \vee s \approx t_2$, a and b be two distinct characters. We can construct a term k to be $\mathsf{con}(t_1, t_2, s, a, t_1, t_2, s, b)$. We denote E

to be a single equality

$$\mathsf{con}(k^2, t_1, k^2, t_2, k^2) \approx \mathsf{con}(x, k^2, s, k^2, y), \tag{2.7}$$

where x and y are two fresh variable of sort String. S and E are equisatisfiable. $\square$

Note that the term k is *primitive* in $\mathcal{A} \cup \mathcal{X}$, where $\mathcal{A}$ is the alphabet and $\mathcal{X}$ is the set of variables. Because of the above propositions, the input of Makanin's algorithm is one single word equality.

In order to solve word equations, the algorithm introduces the system of boundary equations. A *boundary equation* is a tuple that describes the property of a variable and its reverse within the system. The algorithm reduces a word equality to a system of boundary equations and claims that a word equality has a solution iff its boundary equations has a solution. In [60], Makanin also proves that it is decidable whether a system of boundary equations has a solution. In addition, it is shown that the smallest space requirement for a full implementation of Makanin's algorithm does not exceed EXPSPACE.

Even since Makanin showed that a pure word equation is decidable [60], although the original algorithm is incomputable, various improvements have been presented. Those improvements are collected in the Handbook of Formal Languages [79, 80, 81] and two volumes of lecture notes that are dedicated to word equations and related topics [83, 3].

Among them, Plandowski devised a new method for solving the satisfiability of a word equation based on data compression, which can be considered as an independent work from Makanin. Plandowski's approach can be divided into several stages.

In [71], he showed the satisfiability problem is NEXPTIME if equation is compressed by the Lempel-Ziv encoding [99] [18]. In this algorithm, the procedure first guesses the length of a minimal encoded solution, and then finds the solution based on this length. This result is based on the conjecture that the length of a minimal encoded solution is at most exponential to the number of symbols in the word equation. Later, he improved that the satisfiability problem of a word equality is PSPACE in [72]. The main contribution of this work is to show that the complexity of Makanin's algorithm can be further lowered down by data compression.

Moreover, the existential theory of word equations can be extended by allowing linear arithmetic constraints over string length terms, namely the *existential theory of word equations with length constraints*. We use T_{SL} to denote this theory. Many security checking applications can be encoded as formulas in this theory, e.g., [82]. We call these formulas T_{SL}-constraints.

As shown in Example 2, strings with the concatenation forms a monoid, integers with the addition forms another monoid, and the length function is a monoid morphism. Even though the satisfiability of linear arithmetic constraints over $\mathbb{N}$ is NP-complete and the satisfiability of word equations is PSPACE, the decidability of the satisfiability over word equations and length constraints is still an open problem [36].

On the side of the language, it is well-known that two regular languages are closed under common operations (e.g., concatenation, union, intersection, com-

[18]This compression algorithm is widely used in file compression and in GIF image format.

plementation); however, the complexity of performing most of these operations is high. For example, the intersection operation of two regular expressions is PSPACE-complete [53]. In [42], Hooimeijer showed that the satisfiability problem of regular membership constraints is decidable. When regular membership constraints are combined with length constraints, to the best of our knowledge, our approach is the first decision procedure for this full theory. Note that our regular expression length rules relate to the Parikh images [70], where a *semi-linear* set of constraints can be generated. With respect to the Parikh images of a regular expression, [8] described a similar Parikh image as the one we used in part of our length reduction, although we noticed their contribution after we made it by ourselves.

The idea of symbolic regular expression is somehow inspired by the works of symbolic automata [91], where Veanes combined SMT with automata by labeling a transition with a formula instead of a single character. However, since the conversion operation from a regular expression to an automaton (and vice versa) is normally very expensive, and under the context of verification and security applications most problems are more natural to be described via regular expressions, it would be better if an algorithm can directly work on regular expressions. Our work fits the demand of such algorithm. Our approach avoids the computation of these expensive conversions and solves symbolic membership constraints directly. Moreover, with the help of the DPLL(T) architecture, our approach can reason about symbolic membership constraints together with other theories.

Proposition 14 The satisfiability of word equations with membership constraints

becomes undecidable if the language is context-free. □

This proposition is given by [57]. Note that our symbolic regular expression can denote a context-sensitive language. This proposition indicates that our full existential theory over strings may not be decidable.

CHAPTER 3

SOLVING STRING AND LENGTH CONSTRAINTS

In this chapter, we present an approach based on algebraic techniques for solving (quantifier-free) constraints natively over a theory of unbounded strings with length. Our techniques can be used to construct string solvers that can be integrated into general, multi-theory SMT solvers based on the DPLL(T) architecture [66]. We have implemented these techniques in our SMT solver CVC4. As a result and to the best of our knowledge, CVC4 is the first solver able to reason about a language of mixed constraints that includes strings together with integers, reals, arrays, and algebraic datatypes. Furthermore, our experimental results show that CVC4 has superior performance and reliability over specialized string solvers that can reason about the same fragment of the theory of strings.

We have published some results about our calculus in [56]. In addition, we provide proofs for the correctness and solution completeness for this calculus. We also extend our calculus for handling common string manipulating functions, such as substr, contains.

Organization. In Section 3.1, we present our calculus for solving unbounded string constraints and additional length constraints. In addition, we describe how we implement our calculus in CVC4. We provide proofs for refutation soundness, solution soundness and solution completeness of our calculus. We also prove that our calculus gives a decision procedure for constraints in, what we call, *acyclic forms*. In

Sections 3.2 and 3.3, we discuss the refinements and extensions of our calculus respectively. In Section 3.4, we present an experimental evaluation of our implementation in CVC4 and compare it against other tools specializing in string constraints. We conclude in Section 3.5.

3.1 Calculus for T_{SL}

In Section 2.6, we introduced the satisfiability problem in string constraints, and we described different existential theories over strings. In this chapter, we are interested in checking the T_{SL} satisfiability of a finite set of T_{SL}-constraints. We are not aware of any positive results on the decidability of this problem. One complication is that the property that two strings have the same length is not existentially definable by word equations [22, 58]. In fact, the decidability of T_{SL}, is classified as an open question by other authors (e.g., [36]).

In this work we focus on practical solvers for T_{SL} that, although incomplete in general (due to non-termination), can be used to efficiently solve string constraints arising from real-life verification and security applications. In addition to the efficiency, we also focus on correctness. We have developed a solver that is both *refutation sound*: if our solver classifies a problem as unsatisfiable, it is indeed so; and *solution sound*: any variable assignment that the solver claims to be a model to the input constraints does indeed satisfy the input constraints.

Our solver is based on the modular combination [1] of an off-the-shelf solver for

[1] More details are described in Section 2.5

linear integer arithmetic and a novel solver for string and membership [2] constraints. The string solver is obtained as a modular extension of a congruence-closure-based solver for EUF, the theory of equality with uninterpreted functions. The extension is obtained by means of theory-specific derivation rules that assert additional string constraints, length constraints and membership constraints to the congruence closure module (which treats all function symbols as uninterpreted). The combination between the string solver and the arithmetic solver is achieved, in Nelson-Oppen style, by exchanging (dis-)equalities over shared terms. These shared terms, however, are not variables, as in traditional combination procedures [63], but are terms of the form $(\mathsf{len}\, x)$ where x is a variable.[3]

In the following, we describe the essence of our combined solver for T_{SL} abstractly and declaratively, as a tableaux-style calculus. Because of the computational complexity of solving word equations alone, this calculus is non-deterministic and allows many possible proof strategies. Our solver can be understood as a specific proof procedure for the calculus. In our description of our proof procedure below, we focus on the derivation rules that deal with string and arithmetic constraints only. Note that although our procedure works on problems without membership constraints, it will introduce membership constraints by some rules. In this chapter we mainly focus on solving word equations with length constraints and membership constraints can be

[2] A decision procedure for regular membership constraints with length constraints is described in Chapter 4.

[3] This difference is not substantial if the arithmetic solver treats $(\mathsf{len}\, x)$ like an integer variable.

processed in a fairly naive way . In particular, the Kleene star operator is processed by unrolling (lazily): $\mathsf{in}(s, \mathsf{star}(R))$ is reduced to

- $s = \epsilon$ or to
- $s \approx \mathsf{con}(x, y) \wedge \mathsf{in}(x, R) \wedge \mathsf{in}(y, \mathsf{star}(R))$

where x and y are fresh variables, and R is a regular expression. Such unrolling in general may make the solver non-terminating over the membership constraints like Kleene star. A more sophisticated processing of membership constraints will be presented in Chapter 4. There, we mainly focus on a decision procedure for regular membership constraints with length constraints.

Definition 1 Let S be a set of string constraints and let $\mathcal{T}(S)$ be the set of all terms (and subterms) occurring in S. The *congruence closure* of S is the set:

$$\begin{aligned}\mathcal{K}(S) = \quad & \{s \approx t \mid s, t \in \mathcal{T}(S),\ S \models s \approx t\} \cup \\ & \{l_1 \not\approx l_2 \mid l_1, l_2 \text{ distinct string const.}\} \cup \\ & \{s \not\approx t \mid s, t \in \mathcal{T}(S),\ s' \not\approx t' \in S,\ S \models s \approx s' \wedge t \approx t' \text{ for some } s', t'\} \quad (3.1)\end{aligned}$$

□

The set $\mathcal{K}(S)$ induces an equivalence relation $\mathbf{E}_S$ over $\mathcal{T}(S)$ where two terms s, t are equivalent iff $s \approx t \in \mathcal{K}(S)$ (or, equivalently, iff $S \models s \approx t$). For all $t \in \mathcal{T}(S)$, we denote its equivalence class in $\mathbf{E}_S$ by $[t]_S$ or just $[t]$ when S is clear.

As in Chapter 2, we will denote characters (i.e., elements of the alphabet $\mathcal{A}$) by letter c and string constants by letter l or the juxtaposition $c_1 \cdots c_n$ of their individual

$$\mathsf{con}(\boldsymbol{s}, c_1 \cdots c_i, c_{i+1} \cdots c_n, \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, c_1 \cdots c_n, \boldsymbol{u}) \tag{3.2}$$

$$\mathsf{len}(\mathsf{con}(s_1, \ldots, s_n)) \to \mathsf{len}\, s_1 + \cdots + \mathsf{len}\, s_n \tag{3.3}$$

$$\mathsf{con}(\boldsymbol{s}, \mathsf{con}(\boldsymbol{t}), \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u}) \tag{3.4}$$

$$\mathsf{con}(\boldsymbol{s}, \epsilon, \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, \boldsymbol{u}) \tag{3.5}$$

$$\mathsf{len}(c_1 \cdots c_n) \to n \tag{3.6}$$

$$\mathsf{con}(s) \to s \tag{3.7}$$

$$\mathsf{con}() \to \epsilon \tag{3.8}$$

Figure 3.1. String term rewriting rules.

characters Notice that $c_1 \cdots c_n$ denote the empty string ϵ when $n = 0$. We will use x, y, z to denote string variables and s, t, u, v, w to denote terms in general.

We will consider term tuples $(s_1, \ldots, s_n)$, with $n \geq 0$, and denote them by letters in bold font, with comma denoting tuple concatenation. For example, if $\boldsymbol{s} = (s_1, s_2)$ and $\boldsymbol{t} = (t_1, t_2, t_3)$ we will write $(\boldsymbol{s}, \boldsymbol{t})$ to denote the tuple $(s_1, s_2, t_1, t_2, t_3)$. Similarly, if u is a term, then $(\boldsymbol{s}, u, \boldsymbol{t})$ denotes the tuple $(s_1, s_2, u, t_1, t_2, t_3)$.

Definition 2 Our calculus operates over *configurations* consisting of the distinguished configuration unsat and of tuples of the form $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ where

- S, A, R are respectively a set of string, arithmetic, and membership constraints;
- F is a set of pairs $s \mapsto \boldsymbol{a}$ where $s \in \mathcal{T}(\mathsf{S})$ and $\boldsymbol{a}$ is a tuple of atomic string terms;
- N is a set of pairs $e \mapsto \boldsymbol{a}$ where e is an equivalence class of $\mathbf{E}_{\mathsf{S}}$, the equivalence relation induced by the constraints in S, and $\boldsymbol{a}$ is a tuple of atomic string terms;

- C is a set of terms of sort String;
- B is a set of *buckets* where each bucket is a set of equivalence classes of $\mathbf{E}_\mathsf{S}$.

□

Informally, the sets S, A, R initially store the input problem and grow with additional constraints derived by the calculus; N stores a *normal form* for each equivalence class in $\mathbf{E}_\mathsf{S}$; F maps selected input terms to an intermediate form, which we call a *flat form*, used to compute the normal forms in N; C stores terms whose flat form should not be computed, to prevent loops in the computation of their equivalence class' normal form; B eventually becomes a partition of $\mathbf{E}_\mathsf{S}$ used to generate a satisfying assignment that assigns string constants of different lengths to variables in different buckets, and different string constants of the *same* length to different variables in the same bucket.

Definition 3 The calculus is defined by the *derivation rules* described below. A *derivation tree* for the calculus is a tree where each node is a configuration and each non-root node is obtained by applying one of the derivation rules to its parent node. We call the root of a derivation tree an *initial* configuration. A branch of a derivation tree is *closed* if it ends with unsat. A derivation tree is *closed* if all of its branches are closed. □

An initial configuration encodes a satisfiability problem by storing it in the components S, A and R. By distributing constraints based on their signatures, one

can convert any finite set of T_{SL}-constraints into an equisatisfiable [4] set $S \cup A \cup R$ where S is a set of string constraints, A is a set of arithmetic constraints, and R is a set of membership constraints.

Assumption 1 We consider only initial configurations where the other components in the tuple are empty. For convenience, we assume that the S component of the initial configuration contains an equation $x \approx t$ for each non-variable term $t \in \mathcal{T}(\mathsf{S})$, where x is a fresh string variable.[5] We also assume that all terms in any configuration are reduced with respect to the rewrite rules in Figure 3.1, which can be shown to be terminating and confluent modulo the axioms of arithmetic. □

Definition 4 We say that a configuration is *derivable* if it occurs in a derivation tree whose initial configuration satisfies Assumption 1. □

We denote by $t{\downarrow}$ the normal form of a term t with respect to the rewrite rules in Figure 3.1. It is not difficult to see that if t is of sort String, then $t{\downarrow}$ is either an atomic string term or has the form $\mathsf{con}(a_1, \ldots, a_n)$ where $n > 1$ and $a_1, \ldots, a_n$ are atomic; if t is of integer sort, then $t{\downarrow}$ is an arithmetic term. In a similar vein, we consider *normalized* tuples $\boldsymbol{a}{\downarrow}$ of atomic terms obtained from an atomic term tuple $\boldsymbol{a}$ by dropping its empty string components and replacing adjacent string constants by the constant corresponding to their concatenation. For example, $(x, \epsilon, c_1, c_2c_3, y){\downarrow} = (x, c_1c_2c_3, y)$.

[4] By equisatisfiability, we mean T_{SL}-constraints are satisfiable iff $S \cup A \cup R$ is satisfiable.

[5] Such equations can always be added as needed using fresh variables without changing the satisfiability of the original problem.

Invariant 1 We are interested in proof procedures that maintain these invariants on the derivable configurations of the form $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$:

1. All terms are reduced with respect to the rewrite system in Figure 3.1.
2. F is a partial map from $\mathcal{T}(\mathsf{S})$ to normalized tuples of atomic terms.
3. N is a partial map from $\mathbf{E}_{\mathsf{S}}$ to normalized tuples of atomic terms.
4. For all terms s where $[s] \mapsto (a_1, \ldots, a_n) \in \mathsf{N}$ or $s \mapsto (a_1, \ldots, a_n) \in \mathsf{F}$, we have (i) $\mathsf{S} \models_{\mathsf{SL}} s \approx \mathsf{con}(a_1, \ldots, a_n)$ and (ii) $\mathsf{S} \models a_i \not\approx \epsilon$ for $i = 1, \ldots, n$.
5. For all $B_1, B_2 \in \mathsf{B}$, $[s] \in B_1$ and $[t] \in B_2$, $\mathsf{S} \models \mathsf{len}\, s \approx \mathsf{len}\, t$ iff $B_1 = B_2$.
6. C contains only reduced terms of the form $\mathsf{con}(\boldsymbol{a})$.

□

Definition 5 We denote by $\mathcal{D}(\mathsf{N})$ the *domain* of the partial map N, i.e., the set

$$\{e \mid e \mapsto \boldsymbol{a} \in \mathsf{N} \text{ for some } \boldsymbol{a}\}. \tag{3.9}$$

□

For all $e \in \mathcal{D}(\mathsf{N})$, we write $\mathsf{N}\, e$ to denote the (unique) tuple associated to e by N. We will use a similar notation for F.

3.1.1 Derivation Rules

The rules of the calculus are provided in Figures 3.2 through 3.6 in *guarded assignment form*. A derivation rule applies to a configuration c if all of the rule's

A-Conflict $\dfrac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\mathsf{unsat}}$

A-Prop $\dfrac{\mathsf{S} \models \mathsf{len}\, x \approx \mathsf{len}\, y}{\mathsf{A} := \mathsf{A}, \mathsf{len}\, x \approx \mathsf{len}\, y}$

S-Prop $\dfrac{\mathsf{A} \models_{\mathsf{LIA}} \mathsf{len}\, x \approx \mathsf{len}\, y}{\mathsf{S} := \mathsf{S}, \mathsf{len}\, x \approx \mathsf{len}\, y}$

Len $\dfrac{x \approx t \in \mathcal{K}(\mathsf{S}) \quad x \in \mathcal{V}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{len}\, x \approx (\mathsf{len}\, t)\!\downarrow}$

Len-Split $\dfrac{x \in \mathcal{V}(\mathsf{S} \cup \mathsf{A}) \quad x : \mathsf{String}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}\, x > 0}$

R-Star $\dfrac{\mathsf{in}(s, \mathsf{star}(\mathsf{set}(t))) \in \mathsf{R} \quad s \not\approx \epsilon \in \mathcal{K}(\mathsf{S})}{\mathsf{S} := \mathsf{S}, (s)\!\downarrow \approx \mathsf{con}(t, z)\!\downarrow \quad \mathsf{R} := \mathsf{R}, \mathsf{in}(z, \mathsf{star}(\mathsf{set}(t)))}$

Figure 3.2. Rules for theory combination, arithmetic and membership constraints. The letter z denotes a fresh Skolem variable.

premises hold for c. A rule's conclusion describes how each component of c is changed, if at all. We write S, t as an abbreviation for $S \cup \{t\}$. Rules with two conclusions, separated by the symbol $\|$, are non-deterministic branching rules.

In the rules of the calculus, we treat a string constant l in a tuple of terms indifferently as term or a tuple $l_1, \ldots, l_n$ of string constants whose concatenation equals l. For example, a tuple $(x, c_1c_2c_3, y)$ with the three-character constant $c_1c_2c_3$ will be seen also as the tuple (x, c_1, c_2c_3, y), (x, c_1c_2, c_3, y), or (x, c_1, c_2, c_3, y).

All equalities and disequalities in the rules are treated modulo symmetry of $\approx$. We assume the availability of a procedure for checking entailment in the theory of linear integer arithmetic ($\models_{\mathsf{LIA}}$) and one for computing congruence closures and checking entailment in EUF ($\models$).

Reset $$\frac{}{\mathsf{F} := \emptyset \quad \mathsf{N} := \emptyset \quad \mathsf{B} := \emptyset}$$

S-Conflict $$\frac{s \approx t \in \mathcal{K}(\mathsf{S}) \quad s \not\approx t \in \mathcal{K}(\mathsf{S})}{\mathsf{unsat}}$$

S-Split $$\frac{x, y \in \mathcal{V}(\mathsf{S}) \quad x \approx y,\ x \not\approx y \notin \mathcal{K}(\mathsf{S})}{\mathsf{S} := \mathsf{S}, x \approx y \quad \| \quad \mathsf{S} := \mathsf{S}, x \not\approx y}$$

S-Cycle $$\frac{\begin{array}{c} t = \mathsf{con}(t_1, \ldots, t_i, \ldots, t_n) \quad t \in \mathcal{T}(\mathsf{S}) \setminus \mathsf{C} \\ t_k \approx \epsilon \in \mathcal{K}(\mathsf{S}) \text{ for all } k \in \{1, \ldots, n\} \setminus \{i\} \end{array}}{\mathsf{S} := \mathsf{S}, t \approx t_i \quad \mathsf{C} := (\mathsf{C}, t) \setminus \{t_i\}}$$

L-Split $$\frac{x, y \in \mathcal{V}(\mathsf{S}) \quad x, y : \mathsf{String} \quad \mathsf{S} \not\models \mathsf{len}\, x \approx \mathsf{len}\, y \quad \mathsf{S} \not\models \mathsf{len}\, x \not\approx \mathsf{len}\, y}{\mathsf{S} := \mathsf{S}, \mathsf{len}\, x \approx \mathsf{len}\, y \quad \| \quad \mathsf{S} := \mathsf{S}, \mathsf{len}\, x \not\approx \mathsf{len}\, y}$$

Figure 3.3. Basic string derivation rules.

When reading the rules it helps to keep in mind that every non-variable string term in a configuration is equated to a variable in the S component. We divide the rules is several groups here to facilitate their description. The first four rules in Figure 3.2 describe the interaction between arithmetic reasoning and string reasoning, achieved via the propagation of entailed constraints in the shared language. Rule **A-Conflict** derives unsat if the arithmetic part of the problems unsatisfiable. **R-Star** is the only rule for handling membership constraints that we provide here. We chose it because the rule **F-Loop** in Figure 3.5 can generate the constraints matching premises of **R-Star**, even when the initial configuration contains no membership constraints. Note that in this chapter, we focus on solving word equations with length constraints, we assume there is no additional membership constraints in an initial configuration. If a membership constraint is added to R, it comes from the application of **F-Loop**,

and it is in the form of $\mathsf{in}(x, \mathsf{star}(\mathsf{set}(t)))$, where t is a term of String. Thus, the rule **R-Star** is enough for handling such constraints. A set of additional rules to handle more sophisticated membership constraints are introduced in Chapter 4. The basic rules for string constraints are shown in Figure 3.3. The functionality and rationale of **S-Conflict**, **S-Split** and **L-Split** should be straightforward. **S-Conflict** derives unsat if $\mathcal{K}(\mathsf{S})$ contains both equality and disequality between the same pair of strings. **S-Split** tries to guess whether two strings are equal or not, while **L-Split** tries to guess whether two string variables have the same length.

Reset is meant to be applied right after the set S changes, since in that case normal and flat forms may need updating. **S-Cycle** shrinks a concatenation term to one sub-term when the remaining ones are all equivalent to ϵ.

The bulk of the work is done by the rules in Figures 3.4 and 3.5. Those in Figure 3.4 compute a flat form (consisting of a sequence of atomic terms) for each non-variable term that is not in the set C. Flat forms are used in turn to compute normal forms as follows: when all terms of an equivalence class e, except for variables and terms in C, have the same flat form, that form is chosen by **N-Form1** as the normal form of e. When an equivalence class e consists only of variables and terms in C, one of them is chosen by **N-Form2** as the normal form of e. The first two rules of Figure 3.5 use flat forms to add new equations (to S) that are entailed by S in the theory of strings. **F-Loop** is used to recognize and break certain occurrences of *loops* that lead to infinite paths in a derivation tree.

The rules in Figure 3.6 are used to put equivalence classes of terms of sort

$$\textbf{F-Form1}\quad \frac{t = \mathsf{con}(t_1, \ldots, t_n) \quad t \in \mathcal{T}(\mathsf{S}) \setminus (\mathcal{D}(\mathsf{F}) \cup \mathsf{C}) \quad \mathsf{N}\,[t_1] = \boldsymbol{s}_1 \quad \cdots \quad \mathsf{N}\,[t_n] = \boldsymbol{s}_n}{\mathsf{F} := \mathsf{F}, t \mapsto (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n)\!\downarrow}$$

$$\textbf{F-Form2}\quad \frac{l \in \mathcal{T}(\mathsf{S}) \setminus \mathcal{D}(\mathsf{F})}{\mathsf{F} := \mathsf{F}, l \mapsto (l)}$$

$$\textbf{N-Form1}\quad \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad s \in [x] \setminus (\mathsf{C} \cup \mathcal{V}(\mathsf{S})) \quad \mathsf{F}\,t = \mathsf{F}\,s \text{ for all } t \in [x] \setminus (\mathsf{C} \cup \mathcal{V}(\mathsf{S}))}{\mathsf{N} := \mathsf{N}, [x] \mapsto \mathsf{F}\,s}$$

$$\textbf{N-Form2}\quad \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad [x] \subseteq \mathsf{C} \cup \mathcal{V}(\mathsf{S})}{\mathsf{N} := \mathsf{N}, [x] \mapsto (x)}$$

Figure 3.4. Normalization derivation rules. The letter l denotes a string constant.

String into buckets based on the expected length of the value they will be given eventually by a satisfying assignment. The main idea is that different equivalence classes go into different buckets (using **D-Base**) unless they have the same length. In the latter case, they go into the same bucket only if we can tell they cannot have the same value (using **D-Add**). **D-Split** is used to reduce the problem to one of the two previous cases. The goal is that, on saturation, each bucket B can be assigned a unique length n_B, and each equivalence class in B can evaluate to a unique string constant of that length. **Card** makes sure that n_B is big enough to have enough string constants over the alphabet $\mathcal{A}$ of length n_B.

3.1.2 Proof Procedure

We describe a proof procedure that is a highly abstracted version of the one we have implemented. The main procedure is based on the repeated application of the calculus rules according to the seven steps defined below (also as shown in Figure 3.7).

$$\textbf{F-Unify}\quad \frac{\mathsf{F}\, s = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{F}\, t = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad s \approx t \in \mathcal{K}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\, u \approx \mathsf{len}\, v}{\mathsf{S} := \mathsf{S}, u \approx v}$$

$$\textbf{F-Split}\quad \frac{\begin{array}{c}\mathsf{F}\, s = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{F}\, t = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad s \approx t \in \mathcal{K}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\, u \not\approx \mathsf{len}\, v \\ u \notin \mathcal{V}(\boldsymbol{v}_1) \quad v \notin \mathcal{V}(\boldsymbol{u}_1)\end{array}}{\mathsf{S} := \mathsf{S}, u \approx \mathsf{con}(v, z) \quad \| \quad \mathsf{S} := \mathsf{S}, v \approx \mathsf{con}(u, z)}$$

$$\textbf{F-Loop}\quad \frac{\mathsf{F}\, s = (\boldsymbol{w}, x, \boldsymbol{u}_1) \quad \mathsf{F}\, t = (\boldsymbol{w}, v, \boldsymbol{v}_1, x, \boldsymbol{v}_2) \quad s \approx t \in \mathcal{K}(\mathsf{S}) \quad x \notin \mathcal{V}(v, \boldsymbol{v}_1)}{\begin{array}{c}\mathsf{S} := \mathsf{S},\ x \approx \mathsf{con}(z_2, z),\ \mathsf{con}(v, \boldsymbol{v}_1)\!\downarrow \approx \mathsf{con}(z_2, z_1),\ \mathsf{con}(\boldsymbol{u}_1)\!\downarrow \approx \mathsf{con}(z_1, z_2, \boldsymbol{v}_2) \\ \mathsf{R} := \mathsf{R}, \mathsf{in}(z, \mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2))) \quad \mathsf{C} := \mathsf{C}, t\end{array}}$$

Figure 3.5. Equality reduction rules. The letters z, z_1, z_2 denote fresh Skolem variables.

When applying a branching rule the procedure tries the left-branch configuration first. It interrupts the current step and restarts as soon as a constraint is added to S. The procedure loops through the steps until it derives a saturated configuration or the unsat one. In the latter case, it continues with another configuration in the derivation tree, if any.

Step 0: Reset: Apply **Reset** to reset buckets, and flat and normal forms.

Step 1: Check for conflicts: Apply **S-Conflict** or **A-Conflict** if the configuration is unsatisfiable due to the current string or arithmetic constraints.

Step 2: Propagate: Propagate entailed equalities between S and A using **S-Prop** and **A-Prop**.

Step 3: Add length constraints: For each non-variable $t \in \mathcal{T}(\mathsf{S})$, apply **Len** to add the equality $\mathsf{len}(x) \approx \mathsf{len}(t)\!\downarrow$ to A. For each variable x in $\mathcal{V}(\mathsf{S} \cup \mathsf{A})$, apply **Len-Split**, and then first exploring the case where $x \approx \epsilon$.

Step 4: Compute Normal Forms for Equivalence Classes. Apply **S-Cycle** to

Card
$$\frac{B \in \mathsf{B} \quad |B| > 1}{\mathsf{A} := \mathsf{A}, len_B > \lfloor \log_{|\mathcal{A}|}(|B| - 1) \rfloor}$$

D-Base
$$\frac{s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{String} \quad \mathsf{S} \models \mathsf{len}\, s \approx len_B \text{ for no } B \in \mathsf{B}}{\mathsf{B} := \mathsf{B}, \{[s]\}}$$

D-Add
$$\frac{\begin{array}{c} s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{String} \quad \mathsf{B} = \mathsf{B}', B \quad \mathsf{S} \models \mathsf{len}\, s \approx len_B \quad [s] \notin B \\ \text{for all } e \in B \text{ there are } \boldsymbol{w}, u, \boldsymbol{u}_1, v, \boldsymbol{v}_1 \text{ such that} \\ (\mathsf{N}\,[s] = (\boldsymbol{w}, u, \boldsymbol{u}_1),\ \mathsf{N}\, e = (\boldsymbol{w}, v, \boldsymbol{v}_1),\ \mathsf{S} \models \mathsf{len}\, u \approx \mathsf{len}\, v,\ u \not\approx v \in \mathcal{K}(\mathsf{S})) \end{array}}{\mathsf{B} := \mathsf{B}', (B \cup \{[s]\})}$$

D-Split
$$\frac{\begin{array}{c} s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{String} \quad B \in \mathsf{B} \quad \mathsf{S} \models \mathsf{len}\, s \approx len_B \quad [s] \notin B \quad e \in B \\ \mathsf{N}\,[s] = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{N}\, e = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad \mathsf{S} \models \mathsf{len}\, u \not\approx \mathsf{len}\, v \end{array}}{\mathsf{S} := \mathsf{S}, u \approx \mathsf{con}(z_1, z_2), \mathsf{len}\, z_1 \approx \mathsf{len}\, v \parallel \mathsf{S} := \mathsf{S}, v \approx \mathsf{con}(z_1, z_2), \mathsf{len}\, z_1 \approx \mathsf{len}\, u}$$

Figure 3.6. Disequality reduction rules, where letters z_1, z_2 denote fresh Skolem variables. For each bucket $B \in \mathsf{B}$, len_B denotes a unique term $(\mathsf{len}\, x)$ where $[x] \in B$. $|\_|$ denotes the cardinality operator.

completion and then the rules in Figure 3.4 to completion. If this does not produce a total map N, there must be some $s \approx t \in \mathcal{K}(\mathsf{S})$ such that $\mathsf{F}\, s$ and $\mathsf{F}\, t$ have respectively the form $(\boldsymbol{w}, u, \boldsymbol{u}_1)$ and $(\boldsymbol{w}, v, \boldsymbol{v}_1)$ with u and v distinct terms. Let x, y be variables with $x \in [u]$ and $y \in [v]$. If S entails neither $\mathsf{len}\, x \approx \mathsf{len}\, y$ nor $\mathsf{len}\, x \approx \mathsf{len}\, y$, apply **L-Split** to them; otherwise, apply any applicable rule from Figure 3.5, giving preference to **F-Unify**.

Step 5: Partition equivalence classes into buckets. First apply **D-Base** and **D-Add** to completion. If this does not make B a partition of $\mathbf{E}_{\mathsf{S}}$, there must be an equivalence class $[x]$ contained in no bucket but such that $S \models \mathsf{len}\, x \approx len_B$ for some bucket B (otherwise **D-Base** would apply). If there is a $[y] \in B$ such that $x \not\approx y \notin \mathcal{K}(\mathsf{S})$, split on $x \approx y$ and $x \not\approx y$ using **S-Split**. Otherwise, let $[y] \in B$

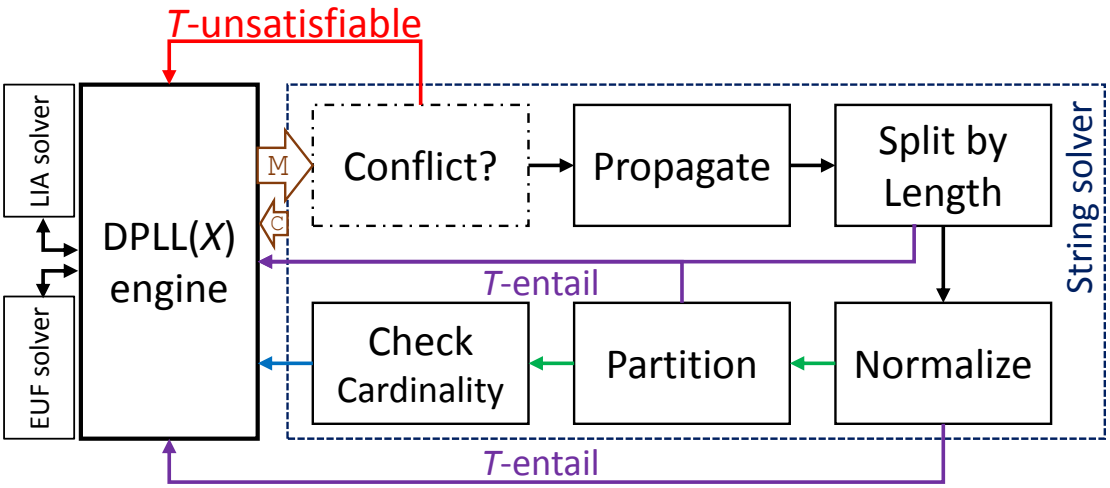

Figure 3.7. Abstracted core proof procedure for strings.

be an equivalence class, such that $x \not\approx y \in \mathcal{K}(\mathsf{S})$. It must be that $\mathsf{N}[x]$ and $\mathsf{N}[y]$ share a prefix followed by two distinct terms u and v. Let x_u, x_v be variables with $x_u \in [u]$ and $x_v \in [v]$. If $S \models \mathsf{len}\, x_u \not\approx \mathsf{len}\, x_v$, apply the rule **D-Split** to u and v. If $S \models \mathsf{len}\, x_u \approx \mathsf{len}\, x_v$, since it is also the case that neither $x_u \approx x_v$ nor $x_u \not\approx x_v$ is in $\mathcal{K}(\mathsf{S})$, apply **S-Split** to x_u and x_v. If S entails neither $\mathsf{len}\, x_u \approx \mathsf{len}\, x_v$ nor $\mathsf{len}\, x_u \not\approx \mathsf{len}\, x_v$, split on them using **L-Split**.

Step 6: Add length constraint for cardinality. Apply the rule **Card** for all buckets B in B, which adds an arithmetic constraint corresponding to the minimal length of terms in B based on the number of equivalence classes in B and the cardinality $|\mathcal{A}|$ of our alphabet.

We prove in Lemma 2 that all derivation trees generated with this proof procedure satisfy Invariant 1. We illustrate the procedure's workings with a couple of examples.

Example 5 Suppose we have the input constraints: $\mathsf{A} = \emptyset$ and $\mathsf{S} = \{\mathsf{len}(x) \approx \mathsf{len}(y), x \not\approx \epsilon, z \not\approx \epsilon, \mathsf{con}(x, l_1, z) \approx \mathsf{con}(y, l_2, z)\}$, where l_1, l_2 are distinct constants of the same length.

After a failure of applying **S-Conflict** and **A-Conflict** for a possible conflict, the proof procedure applies **Len** and **Len-Split** to completion. All resulting derivation tree branches except one can be closed with **S-Conflict**. In the leaf of the non-closed branch every string variable is in a disequality with ϵ. In that configuration, the string equivalence classes are $\{x\}$, $\{y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, and $\{\mathsf{con}(x, l_1, z), \mathsf{con}(y, l_2, z)\}$. The normal form for the first three classes is computed with **N-Form2**; the normal form for the other three with **F-Form2** and **N-Form1**. For the last equivalence class, the procedure uses **F-Form1** to construct the flat forms $\mathsf{F}\,\mathsf{con}(x, l_1, z) = (x, l_1, z)$ and $\mathsf{F}\,\mathsf{con}(y, l_2, z) = (y, l_2, z)$. Then **F-Unify** is applied to add the equality $x \approx y$ to S. After this, the procedure restarts but now with the string equivalence classes $\{x, y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, and $\{\mathsf{con}(x, l_1, z), \mathsf{con}(y, l_2, z)\}$. After similar steps as before, the terms in the last equivalence class get the flat forms (x, l_1, z) and (x, l_2, z) respectively (assuming, without loss of generality, x is chosen as the representative term for $\{x, y\}$). Using **F-Unify** again, the procedure adds the equality $l_1 \approx l_2$ to S and then derives unsat with **S-Conflict**. This closes the derivation tree, showing that the input constraints are unsatisfiable. □

Example 6 Suppose now the input constraints are $\mathsf{A} = \emptyset$ and $\mathsf{S} = \{\mathsf{len}\, x \approx \mathsf{len}\, y,\ x \not\approx \epsilon,\ z \not\approx \epsilon,\ \mathsf{con}(x, l_1, z) \not\approx \mathsf{con}(y, l_2, z)\}$ with l_1, l_2 are distinct constants of the same length.

After similar steps as in Example 5, the procedure can derive a configuration where the string equivalence classes are $\{x\}$, $\{y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, $\{\mathsf{con}(x, l_1, z)\}$, and $\{\mathsf{con}(y, l_2, z)\}$. After computing normal forms for these classes, it attempts to construct a partition B of them into buckets. However, notice that if it adds $\{[x]\}$, say, to B using **D-Base**, then neither **D-Base** (since $S \models \mathsf{len}\, x \approx \mathsf{len}\, y$) nor **D-Add** (since $x \not\approx y \notin \mathcal{K}(\mathsf{S})$) is applicable to $[y]$. So it applies **S-Split** to x and y. In the branch where $x \approx y$, the procedure subsequently restarts, and computes normal forms as usual. At that point it succeeds in making B a partition of the string equivalence classes, by placing $[\mathsf{con}(x, l_1, z)]$ and $[\mathsf{con}(y, l_2, z)]$ into the same bucket using **D-Add**, which applies because their corresponding normal forms are (x, l_1, z) and (x, l_2, z) respectively. Any further rule applications lead to branches with a saturated configuration, each of which indicates that the input constraints are satisfiable. □

Example 7 Suppose now the input constraints are $\mathsf{A} = \emptyset$ and $\mathsf{S} = \{\mathsf{con}(x, a) \approx \mathsf{con}(b, x)\}$ where a, b are distinct constants of length 1.

After applying the rule **F-Loop**, two fresh variables y and z are introduced, and the constraints $a \approx \mathsf{con}(y, z)$ and $b \approx \mathsf{con}(z, y)$ are asserted to S. These two constraints already generate a conflict in S. □

3.1.3 Soundness

We now formalize the main correctness properties of our calculus. Since our solver can be seen as a specific proof procedure, it immediately inherits those properties (*refutation soundness* and *solution soundness*). This means in particular that when our solver terminates with a sat or unsat output, that output is correct. The

correctness arguments in this section are for word equations combined with length constraints. We start with the following lemmas.

Lemma 1 For all terms t of the sort String, $\models_{\mathsf{SL}} t \approx t{\downarrow}$.

Proof: Immediate consequence of the fact that in each of the rewrite rules of Figure 3.1, the left-hand side is equivalent in T_{SL} to the right hand side. ■

Lemma 2 Invariant 1 holds for all derivable configurations.

Proof: First, Invariant 1 trivially holds for any initial configuration by Assumption 1. Thus, we show that parts 1 through 5 of Invariant 1 are preserved for each rule application in a derivation tree.

Part 1 is preserved since all rules only introduce new terms that are normalized with respect to Figure 3.1.

Parts 2 and 3 are only related to rules **F-Form1**, **F-Form2**, **N-Form1** and **N-Form2**. Parts 2 and 3 are preserved since the premises in Figure 3.4 ensure that mappings can be added to F only for terms from $\mathcal{T}(\mathsf{S})$ not in the domain of F, and similarly for N.

Now we show the rules in Figure 3.4 preserve part 4 by cases. Parts 4 is related to rules **F-Form1**, **F-Form2**, **N-Form1** and **N-Form2**. It is preserved for **F-Form1**, since by assumption of part 4 of the invariant on the premises we have that $\mathsf{S} \models_{\mathsf{SL}} t_1 \approx \mathsf{con}(\boldsymbol{s}_1) \wedge \ldots \wedge t_n \approx \mathsf{con}(\boldsymbol{s}_n)$, also con is associative and due to Lemma 1, we have $\mathsf{S} \models_{\mathsf{SL}} t \approx \mathsf{con}(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n){\downarrow}$. It is preserved for **F-Form2**, since $\mathsf{S} \models_{\mathsf{SL}} l \approx \mathsf{con}(l){\downarrow}$ for any l, and $(l){\downarrow}$ is either the empty tuple or a tuple containing a

non-empty string constant. It is preserved for **N-Form1**, since by assumption of the invariant on the premises we have $\mathsf{S} \models_{\mathsf{SL}} s \approx \mathsf{con}(\mathsf{F}s)$, also $x \approx s \in \mathcal{K}(S)$, and thus $\mathsf{S} \models_{\mathsf{SL}} x \approx \mathsf{con}(\mathsf{F}s)$. Finally, it is preserved for **N-Form2**, since $\mathsf{S} \models_{\mathsf{SL}} x \approx \mathsf{con}(x)$ and $\mathsf{S} \not\models x \approx \epsilon$ (as if this was the case, then $[x]$ would contain the term ϵ, which is not in $\mathsf{C} \cup \mathcal{V}(\mathsf{S})$ by assumption of part 6 of the invariant).

Part 5 is related to rules **D-Base** and **D-Add**. Part 5 is preserved by the rule **D-Base**, which creates a new bucket containing equivalence class $[s]$ only when $\mathsf{S} \not\models \mathsf{len}\, s \approx len_B$ for any $B \in \mathsf{B}$. It is also preserved by **D-Add**, which only adds equivalence classes $[s]$ to buckets B when $\mathsf{len}\, s$ is equal to len_B, by assumption of Invariant 1 part 5 on all other $[t] \in B$ and transitivity of equality, we have that $\mathsf{S} \models \mathsf{len}\, s \approx \mathsf{len}\, t$.

Finally, part 6 is preserved by **S-Cycle**. Part 6 is related to rules **S-Cycle** and **F-Loop**. Applying **F-Loop** also preserves part 6, since $\mathsf{F}\, t$ contains a variable x, and thus must have been constructed from an application of **F-Form1**, implying that t is a term of the form $\mathsf{con}(t_1, \ldots, t_n)$. ■

Lemma 3 For all string constants l, l_1, l_2 such that $l \cdot l_2 = l_1 \cdot l$, $l_1 \neq \epsilon$, and $l_2 \neq \epsilon$, there exist string constants k, k_1, k_2 such that $l_1 = k_2 \cdot k_1$, $l_2 = k_1 \cdot k_2$, $l = k_2 \cdot k$, and $k \in (k_1 \cdot k_2)^*$.

Proof: Assume without loss of generality that $l_1 = k_2 \cdot k_1$ for some constants k_1, k_2 where $|k_2| = |l| \bmod |l_1|$. Since $|k_2| < |l_1|$, such k_1 and k_2 always exist. This means $|k_2| = |l| - n \times |l_1|$ for some integer $n \geq 0$, and thus we have $|l| = n \times |l_1| + |k_2|$. We prove the lemma holds for all l such that $l \cdot l_2 = k_2 \cdot k_1 \cdot l$, by induction on n.

If $n = 0$, we have that $|l| = |k_2|$, and thus $l = k_2 \cdot \epsilon$, $l_2 = k_1 \cdot k_2$, and $\epsilon \in (k_1 \cdot k_2)^*$. For $n > 0$, let $l = l' \cdot l''$, where $|l'| = |l_1|$ and $|l''| = (n-1) \times |l_1| + |k_2|$.

By expanding our assumption, we have that $l' \cdot l'' \cdot l_2 = k_2 \cdot k_1 \cdot l' \cdot l''$. Since $|l'| = |l_1|$, we have that $l' = k_2 \cdot k_1$, and $l'' \cdot l_2 = k_2 \cdot k_1 \cdot l''$. Since $|l''| = (n-1) \times |l_1| + |k_2|$, by the induction hypothesis we have that $l_2 = k_1 \cdot k_2$, $l'' = k_2 \cdot k$, and $k \in (k_1 \cdot k_2)^*$ for some k. Due to the length of l'', we have that $l'' = k_2 \cdot (k_1 \cdot k_2)^{(n-1)}$.

Thus, we have $l = k_2 \cdot k_1 \cdot k_2, (k_1 \cdot k_2)^{(n-1)} = k_2 \cdot k'$, where $k' = (k_1 \cdot k_2)^n$ and it is in $(k_1 \cdot k_2)^*$. ■

Lemma 4 If $\langle \mathsf{S}', \mathsf{A}, \mathsf{R}', \mathsf{F}, \mathsf{N}, \mathsf{C}', \mathsf{B} \rangle$ is the result of the application of **F-Loop** to the configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$, then $\mathsf{S}' \cup \mathsf{A} \cup \mathsf{R}'$ is equisatisfiable modulo T_{SL} to $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$.

Proof: Let s, t be terms, u, v be atomic terms, $\boldsymbol{u}_1, \boldsymbol{u}'_1, \boldsymbol{v}_1, \boldsymbol{v}_2$ be vectors of atomic terms, as in the rule **F-Loop**. It is enough to show that the conclusion of **F-Loop** is entailed by S.

Since $s \approx t \in \mathcal{K}(\mathsf{S})$ and due to part 4 of Invariant 1, we have that $\mathsf{S} \models_{\mathsf{SL}} \mathsf{con}(x, \boldsymbol{u}_1) \approx \mathsf{con}(v, \boldsymbol{v}_1, x, \boldsymbol{v}_2)$. Thus, $\mathsf{S} \models_{\mathsf{SL}} \mathsf{con}(\boldsymbol{u}_1) \approx \mathsf{con}(\boldsymbol{u}'_1, \boldsymbol{v}_2)$, and $\mathsf{S} \models_{\mathsf{SL}} \mathsf{con}(x, \boldsymbol{u}'_1) \approx \mathsf{con}(v, \boldsymbol{v}_1, x)$ for some $\boldsymbol{u}'_1$.

Due to part 4 of Invariant 1, we have that $\mathsf{S} \models_{\mathsf{SL}} \mathsf{con}(v, \boldsymbol{v}_1) \not\approx \epsilon$. Due to Lemma 3, we have that for every model of S, there exist k, k_1, k_2, such that the following constraints are true in the model: $\mathsf{con}(v, \boldsymbol{v}_1) \approx \mathsf{con}(k_2, k_1)$, $\mathsf{con}(\boldsymbol{u}'_1) \approx \mathsf{con}(k_1, k_2)$, $x \approx \mathsf{con}(k_2, k)$, and an additional membership constraint $\mathsf{in}(k, \mathsf{star}(\mathsf{set}(\mathsf{con}(k_1, k_2))))$.

Thus, we have that S entails $\mathsf{con}(v, \boldsymbol{v}_1) \approx \mathsf{con}(k_2, k_1)$, $\mathsf{con}(\boldsymbol{u}_1) \approx \mathsf{con}(k_1, k_2, \boldsymbol{v}_2)$, $x \approx \mathsf{con}(k_2, k)$, and $k \in \mathsf{star}(\mathsf{con}(\mathsf{set}\, k_1, \mathsf{set}\, k_2))$ for fresh variables k, k_1, k_2. Thus, the lemma holds. ■

Now we show that our calculus is both refutation sound and solution sound.

Theorem 1 (Refutation Soundness for T_{SL}) *For any closed derivation tree rooted by an initial configuration $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, the set $S_0 \cup A_0 \cup R_0$ is unsatisfiable in T_{SL}.*

Proof: Assume that $c_0 = \langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ has a closed derivation tree D. We show the theorem by induction on the depth of the derivation tree for all nodes $c = \langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B}\rangle$ in D.

Base Case: if the derivation tree is of depth 1, it has to be an application of **S-Conflict** or **A-Conflict**, then $S_0 \cup A_0 \cup R_0$ is unsatisfiable in T_{SL}.

Induction Hypothesis: for every closed derivation tree of depth n (or less), if it starts with a configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B}\rangle$, then $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SL}.

Step Case: Assume a closed derivation tree is of depth $n + 1$, we show the refutation soundness by analyzing cases when a rule is applied to a root configuration c_0.

Note that the children of c_0, are closed trees. Without loss of generality, say a child closed tree is with the root configuration $\langle \mathsf{S}', \mathsf{A}', \mathsf{R}', \mathsf{F}', \mathsf{N}', \mathsf{C}', \mathsf{B}'\rangle$. By the induction hypothesis, since $\langle \mathsf{S}', \mathsf{A}', \mathsf{R}', \mathsf{F}', \mathsf{N}', \mathsf{C}', \mathsf{B}'\rangle$ is the root of a closed derivation tree, $\mathsf{S}' \cup \mathsf{A}' \cup \mathsf{R}'$ is unsatisfiable in T_{SL}. Thus, to show that $S_0 \cup A_0 \cup R_0$ is unsatisfiable

in T_{SL}, it is enough to show that every rule application preserves the models of the premise configuration.

If the children of c_0 are obtained by an application of **Len-Split**, **S-Split**, or **L-Split** (rules that partition search space by the polarity of a literal), then each child is the root of a closed derivation tree. If we consider the new constraints introduced by each of these rules, they are tautologies. Thus, all models are preserved.

If the children of c_0 are obtained by an application of **Reset**, **N-Form1**, **N-Form2**, **F-Form1**, **F-Form2**, **D-Base** or **D-Add** (rules that do not modify S, A, or R), the property holds trivially.

If the child of c_0 is an application of **Card**, we consider two cases:

- if $|B|$ is less than or equal to $|\mathcal{A}|^{len_B}$, all models are preserved, since the new constraint will not generate any conflict.
- if $|B|$ is greater than $|\mathcal{A}|^{len_B}$, it immediately conflicts with the theory of strings.

Thus, we have that $S_0 \cup A_0 \cup R_0$ is unsatisfiable in T_{SL}.

If the child of c_0 is an application of **Len**, **R-Star**, **A-Prop**, **S-Prop**, **S-Cycle**, **F-Unify** or **F-Loop**, then it is a configuration of the form $\langle \mathsf{S}', \mathsf{A}', \mathsf{R}', \mathsf{F}', \mathsf{N}', \mathsf{C}', \mathsf{B}' \rangle$ where $\mathsf{S}' \cup \mathsf{A}' \cup \mathsf{R}'$ is equisatisfiable modulo T_{SL} to $S_0 \cup A_0 \cup R_0$. For **Len**, all satisfying models are preserved because of T_{SL}. For **A-Prop**, and **S-Prop**, this is immediate. For **R-Star**, this holds since s is not empty and s is in $\mathsf{star}(\mathsf{set}(t))$, and thus s must be the concatenation of one or more copies of t. For **S-Cycle**, notice that $S_0 \models_{\mathsf{SL}} t \approx \mathsf{con}(\epsilon, \ldots, t_i, \ldots, \epsilon)$, and hence $S_0 \models_{\mathsf{SL}} t \approx t_i$. For **F-Unify**, we have that by part 4 of

Invariant 1 for s and t, $S_0 \models_{\mathsf{SL}} s \approx \mathsf{con}(\mathsf{F}\, s) \wedge t \approx \mathsf{con}(\mathsf{F}\, t)$, and since $s \approx t \in \mathcal{K}(S_0)$, we have that $S_0 \models_{\mathsf{SL}} \mathsf{con}(\mathsf{F}\, s) \approx \mathsf{con}(\mathsf{F}\, t)$. Thus, since $S_0 \models \mathsf{len}\, u \approx \mathsf{len}\, v$, we have that $S_0 \models_{\mathsf{SL}} u \approx v$. For **F-Loop**, we refer to Lemma 4.

If the children of c_0 are obtained by an application of **F-Split**, then each child is the root of a closed derivation tree. By part 4 of Invariant 1 for s and t, we have that s and t are non-empty strings. Since the lengths of u and v are entailed to be disequal by S, in all models of the original configuration, we have that either u is a prefix of v or vice versa. Given two string variables with different length, all models containing those two variables must fall in one of the branch. This is also proved by Proposition 4.

In addition, we can prove the equisatisfiability between the original configuration and the disjunction of children configuration. If there is a model $\mathcal{M}$ of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ where u is a prefix of v, say $\mathcal{M}[\![u]\!] = l_1$ and $\mathcal{M}[\![v]\!] = l_1 l_2$ (for some l_1, l_2), we can construct an extension of this model $\mathcal{M}'$ such that $\mathcal{M}'[\![k]\!] = l_2$, and $\mathcal{M}'[\![x]\!] = \mathcal{M}[\![x]\!]$ for all other variables x. We have that $\mathcal{M}'$ is a model for $\mathsf{S} \cup \{u \approx \mathsf{con}(v, k)\} \cup \mathsf{A} \cup \mathsf{R}$. A similar construction can be done if there is a model where v is a prefix of u.

If the children of c_0 are obtained by an application of **D-Split**, then each child node is a root of a closed derivation tree. Since the lengths of u and v are entailed to be disequal, in all models of $S_0 \cup A_0 \cup R_0$, either u is longer than v or vice versa. Using similar reasoning as for **F-Split**, by the induction hypothesis on the left branch, we have that there are no models of $S_0 \cup A_0 \cup R_0$ where u is longer than v.

Therefore, every rule application preserves the models of the premise configuration, and the calculus is refutation sound. ■

Definition 6 We say a configuration c is *saturated* with respect to a rule $\mathbf{R}$ if either $\mathbf{R}$ does not apply to it, or all applications of $\mathbf{R}$ to c leave it *unchanged modulo renaming of Skolem variables.* □

A configuration c_1 is called *unchanged modulo renaming of Skolem variables* if adding a new constraint (by applying a rule $\mathbf{R}$) with a set of new free constants V_1, there exists another set of free constants V_2 such that V_1 is a *variant* of V_2. In other words, for each free constant k_1 in V_1, there exists a free constant k_2 in V_2 such that those two free constants can be bijectively renamed.

Definition 7 A derivable configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ is *saturated*, if

- N is a total map over $\mathbf{E}_\mathsf{S}$,
- B is a partition of $\mathbf{E}_\mathsf{S}$, and
- it is saturated with respect to all rules other than **Reset**.

□

Theorem 2 (Solution Soundness for T_{SL}) *If a derivation tree with initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ contains a saturated configuration then the set $S_0 \cup A_0$ is satisfiable in T_{SL}. The saturated configuration induces a satisfying assignment for the set $S_0 \cup A_0$ in T_{SL}.*

Proof: Assume there exists a derivation tree with root node $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ containing a saturated configuration $\langle S_n, A_n, R_n, F_n, N_n, C_n, B_n \rangle$. We will show that we can build a model $\mathcal{M}$ of T_{SRL} that satisfies $S_n \cup A_n \cup R_n$ (which is by construction a superset of $S_0 \cup A_0$). Since all models of T_{SRL} interpret its function and predicate symbols in the same way, to build $\mathcal{M}$ we only need to define its interpretation of the variables in $S_n \cup A_n \cup R_n$.

Before assigning values to variables of type String, we know that the configuration is *saturated*. Thus, there is a model for A_n, say $\mathcal{I}$. We take this model from A_n, and extend it with the rest of constraints. Since the configuration is *saturated*, we can assume these values satisfy the following conditions:

1. $\mathcal{I}[\![t]\!] \bowtie \mathcal{I}[\![s]\!]$ for all $t \bowtie s \in A_n$, where $\bowtie$ is one of $\approx, >$ and t, s are some terms of sort Int;

2. $\mathcal{I}[\![\mathsf{len}\, x]\!] = \mathcal{I}[\![\mathsf{len}\, y]\!]$ if and only if $\mathsf{len}\, x \approx \mathsf{len}\, y \in \mathcal{K}(S_n)$.

For every variable x of type Int in $\mathcal{V}(A_n)$, we build $\mathcal{M}[\![x]\!] = \mathcal{I}[\![x]\!]$.

Due to our selection of these assignments, and part 5 of Invariant 1, we have that $\mathcal{I}[\![len_{B_1}]\!] \neq \mathcal{I}[\![len_{B_2}]\!]$ for all pairs of distinct buckets $B_1, B_2 \in B_n$.

We sort all buckets in B_n to obtain the list $B_{i_1}, \ldots, B_{i_k}$ such that $\mathcal{I}[\![len_{B_{i_1}}]\!] < \ldots < \mathcal{I}[\![len_{B_{i_k}}]\!]$. For $i = i_1, \ldots, i_k$ starting from i_1, we assign values to all variables occurring in the equivalence classes of B_i as followings: model generation is done incrementally with respect to the length of string variables. We say a string constant l is *unused in* $\mathcal{M}$ if we have not assigned $\mathcal{M}[\![x]\!] = l$ for any variable x.

Invariant 2 During model generation, we maintain the invariant:

$$\mathcal{M}[\![\mathsf{len}\, x]\!] = \mathcal{I}[\![\mathsf{len}\, x]\!], \text{ for all variables } x \in \mathcal{V}(S_n). \tag{3.10}$$

□

First, consider equivalence classes $[x] \in B_i$ such that $N_n\,[x]$ is a tuple of the form $(a_1, \ldots, a_n)$, where $[x] \mapsto (a_1, \ldots, a_m)$ was added to N_n by an application of **N-Form1**, say for $F_n\, s = (a_1, \ldots, a_n)$ for some term $s \in [x]$. Since our configuration is *saturated*, it is saturated with respect to the rule **Len** as well. Also, since $x \approx s \in \mathcal{K}(S_n)$, we have that $\mathcal{I}[\![a_{j_1}]\!] + \ldots + \mathcal{I}[\![a_{j_m}]\!] + k = \mathcal{I}[\![len_{B_i}]\!]$, where $\{a_{j_1}, \ldots, a_{j_m}\}$ are the variables in $\{a_1, \ldots, a_n\}$, k is the sum of the lengths of the constants in $\{a_1, \ldots, a_n\}$. Due to our construction of $\mathcal{I}$, part 4(ii) of Invariant 1, and since our configuration is saturated with respect to **Len-Split** as well, we have that $\mathcal{I}[\![a_{j_i}]\!] > 0, \forall i \in [1, m]$. As a result, we have already assigned the value of $\mathcal{M}[\![\mathsf{con}(N_n\,[x])]\!]$ (denoted as l) because:

- if $m = 0$, then l is the empty string;
- otherwise, if $m > 1$, each of a_i is either a string constant or a variable such that $\mathcal{I}[\![\mathsf{len}\, a_i]\!] < \mathcal{I}[\![len_{B_i}]\!]$, in either case a_i has been assigned a value in $\mathcal{M}$ due to the order in which buckets are processed.

Thus, we say $\mathcal{M}[\![y]\!] = l$ for each variable $y \in [x]$. By Invariant 2, for each of $a_{j_1}, \ldots, a_{j_m}$, we have that Formula 3.10 is satisfied for y as well. Since our configuration is saturated with respect to the rule **Card** too, either $|B_i| = 1$, or the value of $\mathcal{I}[\![len_{B_i}]\!]$ is at least $\lfloor \log_{|\mathcal{A}|}(|B_i| - 1) \rfloor + 1$. In either case, there exist at least $|B_i|$

string constants of length $\mathcal{I}[\![len_{B_i}]\!]$. Thus, for all other equivalence classes in B_i (the equivalence classes whose normal form is a variable, say x and thus we denote the corresponding equivalence class as $[x]$), we may choose a string constant l that is unused in $\mathcal{M}$, and say $\mathcal{M}[\![y]\!] = l$ for each variable $y \in [x]$, which clearly satisfies Formula 3.10 as well.

We now argue that $\mathcal{M}$ satisfies $S_n \cup A_n \cup R_n$. Due to Formula 3.10 and by construction of $\mathcal{M}$ for variables of sort Int in $\mathcal{V}(\mathsf{A})$, we have that $\mathcal{M}[\![s]\!] = \mathcal{I}[\![s]\!]$ for all terms $s \in \mathcal{D}(\mathcal{I})$. Thus, due to condition 1 of our construction of $\mathcal{I}$, we have that $\mathcal{M}$ satisfies A_n. Furthermore, since our configuration is saturated with respect to the rule **S-Prop**, $\mathcal{M}$ satisfies all equalities between terms of sort Int in S_n. Due to condition 2 of our construction of $\mathcal{I}$ and since neither **S-Conflict** nor **A-Conflict** applies, $\mathcal{M}$ satisfies all disequalities between terms of sort Int in S_n as well.

Since our configuration is saturated with respect to the rule **R-Star**, we have that $s \approx \epsilon \in \mathcal{K}(S_n)$ for all constraints of the form $\mathsf{in}(s, \mathsf{star}(\mathsf{set}(t))) \in R_n$. Moreover, all constraints in R_n (those introduced by applications of **F-Loop**) are of the form $\mathsf{in}(s, \mathsf{star}(\mathsf{set}(t)))$ for some strings s and t. Since $\mathsf{in}(\epsilon, \mathsf{star}(\mathsf{set}(t)))$ is valid in T_{SRL} for any t, we have that $\mathcal{M}$ satisfies R_n.

To show that $\mathcal{M}$ satisfies the equalities between string terms in S_n, we begin with an intermediate proposition.

Proposition 15 $\mathcal{M}[\![t]\!] = \mathcal{M}[\![N_n[t]]\!]$ for all terms t of sort String in $\mathcal{T}(S_n)$.

Proof: We prove this property by induction on the length of t. The base case holds trivially. We have our induction hypothesis that the lemma holds for all s where

$\mathcal{M}[\![\mathsf{len}\, s]\!] < \mathcal{M}[\![\mathsf{len}\, t]\!]$, and thus $\mathcal{M}$ satisfies all equalities in $\mathcal{K}(S_n)$ between terms whose lengths in $\mathcal{M}$ are strictly less than $\mathcal{M}[\![\mathsf{len}\, t]\!]$. Now we show it also holds for t case by case.

If t is a variable, the statement holds by our construction of $\mathcal{M}$.

If t is not a variable, then by the premises of **N-Form1** and **N-Form2**, either $F_n\, t = N_n\, t$, or $t \in C_n$.

When $F_n\, t = N_n\, t$ and t is a string constant l, then $N_n\, t = (l)\!\downarrow$, and the statement is immediate.

When $F_n\, t = N_n\, t$ and t is a term of the form $\mathsf{con}(t_1, \ldots, t_m)$ for $m > 1$, then $F_n\, t = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$, where $N_n\, t_i = \boldsymbol{s}_i$, $\forall i \in [1, m]$. Since $m > 1$, our configuration is saturated with respect to the rule **Len**, and part 4(ii) of Invariant 1, we have that $\mathcal{M}[\![\mathsf{len}\, t_i]\!] < \mathcal{M}[\![\mathsf{len}\, t]\!]$, $\forall i \in [1, m]$. Thus, by our hypothesis $\mathcal{M}[\![t_i]\!] = \mathcal{M}[\![\boldsymbol{s}_i]\!]$, $\forall i \in [1, m]$. Thus, $\mathcal{M}[\![t]\!] = \mathcal{M}[\![\mathsf{con}(\mathcal{M}[\![\boldsymbol{s}_1]\!], \ldots, \mathcal{M}[\![\boldsymbol{s}_m]\!]]\!] = \mathcal{M}[\![N_n\, [t]]\!]$. We have thus shown the lemma holds for each term $t \notin C_n$.

To show the lemma holds for each term $t \in C_n$, we first identify a *parent term* s distinct from t such that $t \approx s \in \mathcal{K}(S_n)$, and $\mathcal{M}[\![t]\!] = \mathcal{M}[\![s]\!]$. We will construct a (partial) function parent from terms to terms, where $\mathsf{parent}(t)$ denotes the parent term of t.

If $t \in C_n$, and t was added by an application of **S-Cycle**, then t is the concatenation of $t_1, \cdots, t_n$ and since constraints are never removed from S, we have that $t_k \approx \epsilon \in \mathcal{K}(S_n)$ for each $k \in \{1, \ldots, n\} \setminus \{i\}$. We have that either $\mathcal{M}[\![\mathsf{len}\, t]\!] = 0$, or by our hypothesis $\mathcal{M}$ satisfies each of $t_k \approx \epsilon$. In each of these cases, we have that

$\mathcal{M}[\![t]\!] = \mathcal{M}[\![t_i]\!]$, and since $t \approx t_i \in \mathcal{K}(S_n)$, we say $\mathsf{parent}(t) = t_i$.

If $t \in C_n$, and t was added by an application of **F-Loop**, then due to part 4(ii) of Invariant 1, we have that $\mathcal{M}[\![\mathsf{len}\, x]\!] < \mathcal{M}[\![\mathsf{len}\, t]\!]$, $\mathcal{M}[\![\mathsf{len}\, \boldsymbol{u}_1]\!] < \mathcal{M}[\![\mathsf{len}\, s]\!] = \mathcal{M}[\![\mathsf{len}\, t]\!]$, as well as $\mathcal{M}[\![\mathsf{len}\, \boldsymbol{v}_1]\!] < \mathcal{M}[\![\mathsf{len}\, t]\!]$. By our hypothesis, we have $\mathcal{M}[\![s]\!] = \mathcal{M}[\![\mathsf{con}(\boldsymbol{w}, k_2, k, k_1, k_2, \boldsymbol{v}_2)]\!]$, and $\mathcal{M}[\![t]\!] = \mathcal{M}[\![\mathsf{con}(\boldsymbol{w}, k_2, k_1, k_2, k, \boldsymbol{v}_2)]\!]$. Since $\mathcal{M}$ satisfies R_n, we have that $\mathcal{M}[\![k]\!]$ is of the form $(\mathcal{M}[\![k_1]\!] \cdot \mathcal{M}[\![k_2]\!])^m$. For all such m, we have that $\mathcal{M}[\![t]\!] = \mathcal{M}[\![s]\!]$, and since $t \approx s \in \mathcal{K}(S_n)$, we say $\mathsf{parent}(t) = s$.

Consider the path from the initial configuration to our saturated configuration. For all t where $\mathsf{parent}(t)$ is defined, $\mathsf{parent}(t)$ was added to S_n after t. Indeed, notice that **S-Cycle** removes $t_i = \mathsf{parent}(t)$ from C, and **F-Loop** guarantees that $s = \mathsf{parent}(t)$ is not in C since it has a flat form. Thus, we have that $\mathsf{parent}^m(t) \neq t$ for all terms t, $m > 0$, where $\mathsf{parent}^m(t)$ (the m-fold application of parent) is defined. For each $t \in C_n$, consider the largest m such that $\mathsf{parent}^m(t)$ is defined, where by the previous observation have that n is finite. We have that $\mathcal{M}[\![t]\!] = \mathcal{M}[\![\mathsf{parent}(t)]\!] = \ldots = \mathcal{M}[\![\mathsf{parent}^m(t)]\!]$, which, since $\mathsf{parent}^m(t) \notin C_n$, is equal to $\mathcal{M}[\![N_n\,[\mathsf{parent}^m(t)]]\!]$. Additionally, we have that $t \approx \mathsf{parent}(t), \ldots, \mathsf{parent}^{m-1}(t) \approx \mathsf{parent}^m(t) \in \mathcal{K}(S_n)$, and thus $[t] = [\mathsf{parent}^n(t)]$. Putting these together, we have $\mathcal{M}[\![t]\!] = \mathcal{M}[\![N_n\,[t]]\!]$. □

Due to this proposition, and since N_n is a total map, $\mathcal{M}$ satisfies all equalities between terms of type String in S_n. To show $\mathcal{M}$ also satisfies the disequalities between string terms in S_n, we show the following proposition.

Proposition 16 $\mathcal{M}[\![\mathsf{con}(N_n\,[x])]\!] \neq \mathcal{M}[\![\mathsf{con}(N_n\,[y])]\!]$ for each pair of distinct equivalence classes $[x], [y]$ in any bucket B.

Proof: We may assume that the proposition holds for all buckets B' where $\mathcal{M}[\![len_{B'}]\!] < \mathcal{M}[\![len_B]\!]$. Since $[x]$ and $[y]$ occur in the same bucket, due to our premises in the rule **D-Add**, $N_n\,[x]$ and $N_n\,[y]$ must be of the form $(\boldsymbol{w}, u, \boldsymbol{u}_1)$ and $(\boldsymbol{w}, v, \boldsymbol{v}_1)$ respectively, where $S_n \models \mathsf{len}\, u \approx \mathsf{len}\, v$ and $u \not\approx v \in \mathcal{K}(S_n)$. Since each of u and v is either a variable or a string constant, if $\boldsymbol{w}, \boldsymbol{u}_1, \boldsymbol{v}_1$ are empty tuples, then the lemma holds by our construction of $\mathcal{M}$. Otherwise, we have that $\mathcal{M}[\![\mathsf{len}\, u]\!] < \mathcal{M}[\![len_B]\!]$ and $\mathcal{M}[\![\mathsf{len}\, v]\!] < \mathcal{M}[\![len_B]\!]$, and moreover since u and v have the same length, then $[u]$ and $[v]$ occur in the same bucket B'. Due to our assumption for bucket B', we have that $\mathcal{M}[\![\mathsf{con}(N_n\,[u])]\!] \neq \mathcal{M}[\![\mathsf{con}(N_n\,[v])]\!]$ and thus $\mathcal{M}[\![\mathsf{con}(N_n\,[x])]\!] \neq \mathcal{M}[\![\mathsf{con}(N_n\,[y])]\!]$ as well. □

Due to part 5 of Invariant 1 and by our construction of $\mathcal{M}$, we have that $\mathcal{M}[\![\mathsf{len}\, N_n\,[x]]\!] = \mathcal{M}[\![\mathsf{len}\, N_n\,[y]]\!]$ if and only if $[x]$ and $[y]$ occur in the same bucket. Thus, by Propositions 15and 16, we have that $\mathcal{M}[\![s]\!] \neq \mathcal{M}[\![t]\!]$ for all pair of terms s, t where s and t reside in distinct equivalence classes of $\mathcal{K}(S_n)$. Since **S-Conflict** does not apply, we have that $\mathcal{M}$ satisfies all the disequalities between terms of type String in S_n as well. This concludes the proof of Theorem 2. ■

3.1.4 Solution Completeness

According to [36], the decidability problem in T_{SL} is still an open question. We do not have a proof that our calculus is refutation complete in T_{SL}. Here, we give a proof for solution completeness, i.e., if a set of constraints is satisfiable, our calculus is able to find a solution. Since our calculus is solution complete but not necessarily refutation complete, it is not guaranteed to terminate if a set of constraints

is unsatisfiable. Note that refutation completeness does not require the termination on satisfiable problems.

In this subsection, we first prove the termination for our calculus on constraints with an upper bound over the length of all input string variables, then we introduce a new *fair* strategy, and finally, we will give a proof for solution completeness under this strategy.

Definition 8 Let S be a set of string constraints. S is called a set of *bounded* string constraints, if it entails a constraint that defines an upper bound for the sum of all input string variables in T_{SL}, i.e.,

$$\Sigma_{x \in \mathcal{V}(S)} \mathsf{len}(x) \ \leq \ n, \tag{3.11}$$

where n is a non-negative integer constant. □

Proposition 17 Let S be a set of *bounded* string constraints. S entails an upper bound constraint $c = \Sigma_{x \in \mathcal{V}(S)} \mathsf{len}(x) \leq n$, for some natural number n. Then, S and $S \cup \{c\}$ are *equisatisfiable*. □

Given a configuration c_n that is derived from an initial configuration c_0 where S_0 is the set *bounded* string constraints in c_0, we build a partial function p that maps an equivalence class to a natural number that represent a *maximum possible length* of the equivalence class.

Since S_0 is the set of *bounded* string constraints, S_0 entails $\Sigma_{x \in \mathcal{V}(\mathsf{S})} |x| \leq n$, for some natural number n. Initially, every input variable has a normal form of itself, and we define $p([x]) = n, \forall x \in \mathcal{V}(S_0)$.

Note that if c_n is *derivable* form c_0, there is a *unique* path p from the node c_0 to the node c_n in the derivation tree. Consider each node along the path p from c_0 in order. When a *fresh* variable is introduced in a configuration, the *maximum possible length* of each old equivalence class containing only variables is defined. In our calculus for T_{SL} (without **F-Loop**), if a *fresh* variable, say k, is added to S_i in c_i along the path p, this variable must be introduced by **F-Split**, and thus a new constraint in the form of $u \approx \mathsf{con}(v, k)$ is added to S_i, where $N_{i-1}\ [u] = (u)$ and $N_{i-1}\ [v] = (v)$, and u and v are non-empty. Since $p([u])$ is defined and k is newly added, we define $p([k]) = p([u]) - 1$.

We also define another function P that maps a configuration to a multi-set containing maximum possible lengths of variables whose equivalence class contains only variables in a configuration, i.e., $P(c) = \{p([x]) \mid [x] \text{ contains only variables}\}$, where c is a configuration. The relation $\prec_s$ between two multi-sets follows the standard definition of an ordered multi-set ordering (over natural numbers), i.e., for two ordered multi-sets [6] s_1 and s_2, we have $s_1 \prec_s s_2$ if

$$\exists 0 \leq i < \max(|s_1|, |s_2|).\ (\forall 0 \leq j < i.\ s_1[j] = s_2[j]) \wedge s_1[i] < s_2[i]. \tag{3.12}$$

Notice that the minimal multi-set over this relation is the empty set, and this relation is well-founded.

We define a function Q that maps a configuration $c = \langle S, A, R, F, N, C, B\rangle$ to the number of pairs of equivalence classes that are neither equal nor disequal, i.e., $Q(c) = |S_Q(c)|$, where $S_Q(c) = \{([s], [t]) \mid S \not\models s \approx t \wedge S \not\models s \not\approx t\}$.

[6] Elements are in descending order.

We define a binary relation $\prec_c$ between two configurations $c_1 = \langle S_1, A_1, R_1, F_1, N_1, C_1, B_1\rangle$ and $c_2 = \langle S_2, A_2, R_2, F_2, N_2, C_2, B_2\rangle$, such that $c_1 \prec_c c_2$ if :

1. $P(S_1) \prec_s P(S_2)$, or
2. $P(S_1) = P(S_2)$ and $Q(S_1) < Q(S_2)$.

In the following, we prove a sequence of lemmas that lead us to the proof of termination for the bounded case.

Lemma 5 Given a configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B}\rangle$, the number of equalities (containing variables) in $\mathcal{K}(\mathsf{S})$ is finite.

Proof: According to Formula 3.1, we are only interested in the equalities in S. Assume that the number of equalities in S is n and the maximum symbols of a term in $\mathcal{T}(\mathsf{S})$ is k. We know that the number of terms in $\mathcal{T}(\mathsf{S})$ is at most $2n \cdot k(k-1)/2$ ($\leq n \cdot k^2$). Therefore, the number of equalities in $\mathcal{K}(\mathsf{S})$ is no more then $n^2 \cdot k^4$ (the square of the number of terms in $\mathcal{T}(\mathsf{S})$). ■

Lemma 6 Given a configuration $c = \langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B}\rangle$, in any derivation tree rooted by c, the normalization rules (**F-Form1**, **F-Form2**, **N-Form1** and **N-Form2**) can only be applied for finitely many times before a reset (or equivalently a change in S).

Proof: Notice that **Reset** is applied immediately after S is changed. By Lemma 5, we know that the number of terms in $\mathcal{T}(\mathsf{S})$ is finite. The normalization rules (in Figure 3.4) only work on terms in $\mathcal{T}(\mathsf{S})$ and produce no equations, so they will not trigger the **Reset** rule. Notice that at any step, applying any of these rules on the

same terms will not generate any not new terms in either N or F. Therefore, F and N will be saturated by the normalization rules in finite steps. ■

Lemma 7 Given a configuration $c = \langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$, in any derivation tree rooted by c, the B-modifying rules (**D-Base** and **D-Add**) can only be applied for finitely many times before a reset (or equivalently a change in S).

Proof: By Lemma 5, we know that the number of equalities in $\mathcal{T}(\mathsf{S})$ is finite, and thus, the number of equivalent classes is finite. Also, at any step, applying any of these rules on the same terms will not generate any not new terms in B. Therefore, the result is immediate. ■

Lemma 8 Given a configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$, in any derivation tree rooted by c, the A-modifying rules (**A-Prop**, **S-Prop**, **Len**, **Len-Split**, **L-Split** and **Card**) can only be applied for finitely many times before a reset (or equivalently a change in S).

Proof: By Lemma 5, we know that the number of equalities in $\mathcal{T}(\mathsf{S})$ is finite, and thus, the number of equivalent classes is finite. Also, we assume that the number of arithmetic equalities in $\mathcal{A}$ is finite. At any step, applying any of these rules on the same terms will not generate any not new terms in A. Therefore, the result is immediate. ■

Definition 9 A derivation tree is *finite* iff

- it is *closed* (i.e., all leaves contain a *closed* configuration), or
- it is *saturated* (i.e., it contains a *saturated* configuration in one leave).

□

Strategy 1 The original proof strategy (see Section 3.1). In addition, we add the following requirements to show that a derivation tree is *finite*:

- we consider the derivation without **F-Loop** [7],
- if a derivation tree is neither *closed* nor *saturated*, a rule is guaranteed to be applied,
- **Reset** only applied each time when S is changed, and
- the derivation stops immediately after a derivation tree is *saturated*.

□

Theorem 3 (Bounded Termination) *With Strategy 1, for any derivation tree with an initial configuration* $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$*, if* $S_0 \cup A_0$ *entails an upper bound for the sum of all input string variables (i.e.,* $\Sigma_{x \in \mathcal{V}(S_0)} \mathsf{len} x \leq n$*, for some natural number* n*), the derivation tree is finite.*

Proof: By Proposition 17, it is equivalent to consider the configuration where the set of arithmetic constraints contains the upper bound constraint. To show the derivation tree is finite, it is enough to show that there is no infinite path in the derivation tree.

Note that we can group our rules into the following categories:

- conflicting rules: **A-Conflict** and **S-Conflict**

[7] Since **F-Loop** is not applicable, **R-Star** will not be applied.

- A-modifying rules: **A-Prop**, **S-Prop**, **Len**, **Len-Split** [8], **L-Split** and **Card**
- normalization rules: **F-Form1**, **F-Form2**, **N-Form1** and **N-Form2**
- S-modifying rules: **S-Split**, **S-Cycle**, **R-Star**, **F-Unify**, **F-Split** and **D-Split**
- B-modifying rules: **D-Base** and **D-Add**
- reset rule: **Reset**.

If any conflicting rule is applied, the derivation is *closed*.

Notice that our measure of a configuration before applying **Reset** and the measure of configurations before applying any S-modifying rules are identical with respect to the relation $\prec_c$. By Lemmas 8, 6, 7, the number of configurations in-between two resets is finite. We will show that the measure of a configuration decreases after any S-modifying rule with respect to the relation $\prec_c$, we show the termination.

Let's consider each of the S-modifying rule. For each S-modifying rule, we show that either our *well-founded* measure (with respect to the relation $\prec_c$) decreases after application, or our measure stays the same but there are only finitely many applications.

If a configuration is an immediate result of applying **S-Split**, the number of pairs of equivalence classes that are neither equal nor disequal is decreased. Thus, the measure of the child configuration is strictly smaller the order of the parent configuration.

---

[8] Although **Len-Split** does add an equality (in the form of $x \approx \epsilon$) to S, the main effect is in A, so we classifies it as an A-modifying rule.

If a configuration is an immediate result of applying **S-Cycle**, the order of new configuration is not changed; however, we can only apply this rule for finitely many times, at most the number of symbols in each equality.

If a configuration is an immediate result of applying **F-Unify**, by Invariant 1, u and v are two terms in different equivalence classes. If $[u]$ and $[v]$ are disequal by $\mathcal{K}(\mathsf{S})$, adding this equality will immediate lead to **S-Conflict** and thus the procedure terminates; otherwise, adding this equality will decrease the number of pairs of equivalence classes that are neither equal nor disequal.

If a configuration is an immediate result of applying **F-Split**, by Invariant 1, u and v are two non-empty atomic terms. Without losing the generality, we can assume they are either a single character or a variable. If they are distinct string constants, it will lead to **S-Conflict** in the next iteration and thus the procedure terminates. If one is a variable x and the other is a character c [9], since the variable is non-empty and the length is not equal to 1, the branch with $c \approx \mathsf{con}(x, k)$ will be closed by a conflict in the arithmetic constraints. In the other branch, an equality $x \approx \mathsf{con}(c, k)$ (where k is a fresh variable) will be added. By adding this equality, the maximum possible length of k is strictly smaller than x, and the normal form of x is not (x) any more (because the concatenation is also in the equivalence class of x), and thus the measure is decreased. If both are variables, in either branch, a fresh variable with strictly less maximum possible length is introduced and one old variable will be

[9] Although the rule can consider a constant string of arbitrary length, it is enough to consider only a character by Levi's theorem.

removed from the multi-set as it will not have the normal form of itself, and thus the measure decreases.

If a configuration is an immediate result of applying **D-Split**, following the similar argument for **F-Split**, in either branch, two fresh variables with less maximum possible lengths are introduced and one old variable will be removed from the multi-set as it will not be the normal form of itself. Thus, the measure of the child configuration is smaller than the measure of the parent configuration.

Therefore, the measure of configuration is decreasing after apply any of S-modifying rules, or the measure stays the same but there are only finitely many applications of a rule. The derivation tree is finite. ■

Corollary 3 (One-Sided Termination) Given a set of equations: $t_1 \approx l_1, \cdots, t_k \approx l_k$, where t_i's are string terms, l_i's are string constants, the derivation tree is finite.

Proof: The proof is straight forward: the constraints entail that $\Sigma_{x \in \mathcal{V}(t_1) \cup \cdots \cup \mathcal{V}(t_k)} \mathsf{len}(x)$ is less or equal to $\Sigma_{i=1}^{k} \mathsf{len}(l_i)$. By Theorem 3, the deduction is terminating. ■

We expand our configuration with one more extra component b, such that b is a natural number, initially assigned to 0. We add two new rules (shown in Figure 3.8) to our calculus. We use J to denote the term : $\Sigma_{x \in \mathcal{V}(S_0)} \mathsf{len}\, x$.

Strategy 2 (Fairness) In addition to Strategy 1, our *fair* strategy is adjusted as follows:

- the proof procedure always starts with applying the rule **J-Start**, and **J-Start** only applies once;

- when a **J**-rule is applied, rules will be applied to the left subtree first;
- **J-Split** is applied in a subtree iff its counterpart (with respect to the application of a **J**-rule) is *closed*.

□

Theorem 4 (Solution Completeness for T_{SL} with Fairness) *With Strategy 2, for any derivation tree with an initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, if the set $S_0 \cup A_0$ is satisfiable in T_{SL}, the derivation tree will be saturated.*

Proof: By the definition of **J-Start** and the assumption, the derivation starts with the upper bound (of J) equal to 0 on the left branch. By Theorem 3, if an upper bound for the sum of the lengths of all input variables is given, the subtree of the derivation is either closed or saturated. If the subtree is saturated, the whole derivation tree is saturation; otherwise, the left subtree is *closed* which triggers the rule **J-Split** by Strategy 2.

Note that in the root configuration of the right subtree, $J > \mathsf{b}$ is asserted to A by **J-Start**. Because the left branch is *closed*, the **J-Split** is applied with a strictly larger larger bound n for J (which is at least b). By Strategy 2, rules will be applied to the left subtree first with a new upper bound $J \leq n + 1$ and updates b to $n + 1$ accordingly. Note that after each application of **J-Split**, the upper bound is strictly increasing. The left subtree of derivation after applying **J-Split** is always finite by Theorem 3.

Since $S_0 \cup A_0$ is satisfiable in T_{SL}, there exists a model M that assigns each

$$\textbf{J-Start}\quad \frac{}{\mathsf{A} := \mathsf{A}, J \leq 0 \quad \| \quad \mathsf{A} := \mathsf{A}, J > 0}$$

$$\textbf{J-Split}\quad \frac{\mathsf{A} \models J > n, n \geq \mathsf{b}}{\mathsf{A} := \mathsf{A}, J \leq n+1 \quad \mathsf{b} := n+1 \quad \| \quad \mathsf{A} := \mathsf{A}, J > n+1}$$

Figure 3.8. Rules for the fair strategy, where J denotes the term : $\Sigma_{x \in \mathcal{V}(S_0)} \mathsf{len}\, x$, S_0 is the initial set of equations, and n is a non-negative integer constant.

variable to a string constant. Let m be the sum of lengths of all these string constants in M. Because of the refutation soundness in Theorem 1, $\mathsf{A} \not\models J > m$ initially.

Assume that there exists a natural number k, such that $k > m$ and $J > k+1$ is added to A in some configuration. Then, in its counterpart configuration c, $J \leq k+1$ will be added to A. By Theorem 3, the subtree of derivation rooted by c is finite. Due to Theorem 1, the subtree is saturated, and thus, the whole derivation tree is saturated.

By Theorem 2, if a derivation tree is saturated, the saturated configuration contains a model. ■

Corollary 4 (Minimal Model) When a derivation terminates with a model in the *fair* strategy, the model is *minimal* with respect to the sum of lengths of the input variables.

Proof: We can prove this property by contradiction. Assume the sum of lengths of the input variables in the returned model is n, the sum of lengths of input variables in the minimal model is m, and $n > m$. There must be an assertion introduced by **J-Split** (in the right branch), saying $J > k$, such that $k \geq m$. When this happens,

we know the left branch is closed, which also means that there is no model for which $\mathcal{A} \models J \leq k$. It contradicts the assumption. ■

3.1.5 Decision Procedure for Constraints in an Acyclic Form

One of the complications in word equations with length constraints is that some variables may occur more than once in an equation. For example, if a variable occurs on the both side of an equality, the rule **F-Loop** may be applied. This rule may introduce a membership constraint with a symbolic expression denoting a context-sensitive language, and thus it is hard to prove the refutation completeness when such a language happens. Therefore, we want to avoid this situation.

One way to avoid this situation is to limit input constraints to an *acyclic form*. The original acyclic form was introduced by [1]. The rationale of the introduction of this form is to recognize a class of constraints without the following situations:

- a variable appears on both sides of an equality, i.e.,

  $$\mathsf{con}(\boldsymbol{u}_1, x, \boldsymbol{u}_2) \approx \mathsf{con}(\boldsymbol{v}_1, x, \boldsymbol{v}_2),$$

  where x is a variable, and $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{v}_1, \boldsymbol{v}_2$ are vectors of atomic terms;

- a variable appears more than once on one side of an equality, i.e.,

  $$\mathsf{con}(\boldsymbol{u}_1, x, \boldsymbol{u}_2, x, \boldsymbol{u}_3) \approx \mathsf{con}(\boldsymbol{v}),$$

  where x is a variable, and $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3, \boldsymbol{v}$ are vectors of atomic terms;

- a constraint in the previous two forms will not be added during a derivation.

Assumption 2 During the preprocessing of constraints, our procedure replaces every disequality, say $s \not\approx t$, with the following constraint:

$$\begin{aligned} & s \approx \mathsf{con}(t, k_1) \ \vee \ t \approx \mathsf{con}(s, k_1) \ \vee \\ & \quad (s \approx \mathsf{con}(k_1, x, k_2) \ \wedge \ t \approx \mathsf{con}(k_1, x', k_3) \ \wedge \\ & \quad \mathsf{len}\ x \approx \mathsf{len}\ x' \ \wedge \ \mathsf{len}\ x' \approx 1 \ \wedge \ x \not\approx x'), \end{aligned} \tag{3.13}$$

where k_1, k_2, k_3, x, x' are fresh string variables. □

By Assumption 2, the rule **D-Split** will not be applied.

First, we introduce our definition for a dependency graph. This definition is associated with our definition of acyclicity.

Definition 10 A *dependency graph* is an undirected multigraph $G = \langle N, E \rangle$, where

- V is a set of *nodes*, and
- E is a multiset of unordered pairs of nodes, called *edges*.

Each node in N is labeled with a variable and associated with a color of either white or black, and each edge in E is labeled with a number and associated with a color of either red or blue.

Every configuration is associated with a dependency graph. Now, we describe how we construct a dependency graph from the set of string constraints S in a configuration. Given a configuration where S_n is the set of string constraints, we use V_n to denote the set of all variables occurred in the string set S_n. Since our calculus never removes any equation from S, we can always partition the current string set

into two disjoint set S_0 and S'_n, where S_0 is the initial string set and S'_n contains all equalities introduced by our calculus. Note that only **F-Unify** and **F-Split** introduce new constraints to S. Since the introduced constraints are either of the form $u \approx v$ or or the form $u \approx \mathsf{con}(v, k)$, for each equality in S'_n, we can always pick a side which is a string variable. Note that u and v cannot be string constants at the same time. We use U_n to denote the set of the picked variables in S'_n. For each variable x in U_n, we add a *black* node with the label x in the graph; for each variable y in $V_n \setminus U_n$, we add a *white* node with the label y. Therefore, each variable represents a unique node colored with either *white* or *black*. For simplicity, we refer x to be the node x in the dependency graph if context is clear. Initially, all nodes are *white*.

We provide every equality (in S) with a unique number, called *index*. If two variables, say x and y, appear on the different sides of an equality with the index i, we add a *red* edge with the label i, denoted as the label $\{\mathsf{red}, \mathsf{i}\}$ in graph; if two variables, say z and w, appear on the same side of an equality with index j, we add a *blue* edge with label j, denoted as the label $\{\mathsf{blue}, \mathsf{j}\}$ in graph. In this definition, we allow self-loops, as well as multi edges between two nodes but with different labels.

Example 8 Let $\{x \overset{1}{\approx} \mathsf{con}(y, z)\}$ be the string constraint set in the initial configuration (with indexes on top of the symbol $\approx$). The dependency graph of the initial configuration is $G = \langle N, E \rangle$, where N contains three white nodes labeled x, y and z, and E contains two red edges with the index 1 (one is between the node x and the node y and the other is between the node x and the node z) and one blue edge with the index 1 between the node y and the node z. □

We now define the notation *compression* over paths in a dependency graph, which will be used for properties.

Definition 11 A path in a dependency graph is called a *compressed* path iff

- the path only contains *red* edges, and
- all indexes (labeled in edges) are *distinct*.

□

Definition 12 A configuration is in the *cyclic form*, iff there exists a looping path in its *dependency graph* where

- the path contains at most one *blue* edge,
- all indexes (labeled in edges) are *distinct*, and
- it contains only *white* nodes.

We call this looping path a *cycle*. If a configuration is not in the *cyclic form*, we call it an *acyclic* configuration. □

Example 9 Given a set of word equalities (with indexes on top of the symbol $\approx$):

- $\mathsf{con}(x, y, a) \overset{1}{\approx} \mathsf{con}(z, b, w)$,
- $x \overset{2}{\approx} y$,
- $z \overset{3}{\approx} w$,

the dependency graph is shown in Figure 3.9. Since there is a *cycle* between the white nodes x and y, the configuration is in the *cyclic form*. □

An equality is called *linear* if no variable appears more than once in the equality. We now prove that given a set of equalities in an initial configuration, if there is a *non-linear* equality, the configuration is *cyclic*.

Lemma 9 If an initial configuration contains a non-linear equality, the configuration is cyclic.

Proof: Note that every node is colored with *white*, initially. We only need to consider two situations: repetitions on one side or on different sides of an equality.

If it contains an equality where a variable appears more than once on the same side of the equality, say $\mathsf{con}(s_1, x, s_2, x, s_3) \approx t$, there must be a blue self-loop on the white node x.

If it contains an equality where a variable appears more than once on different sides of the equality, say $\mathsf{con}(s_1, x, s_2) \approx \mathsf{con}(t_1, x, t_2)$, there must be a red self-loop on the white node x.

In either case, the configuration is *cyclic*. ■

Note that the acyclicity definition is an *under-approximation* of the whole decidable fragment for our calculus. Our calculus can find an assignment (or a conflict) when a given problem is in the *cyclic* form, e.g., the conflict in Example 9.

In our calculus, no constraints are removed from the string set S, and only **F-Split** and **F-Unify** append S. If **F-Split** detects a pair of *splittable* variables, say u

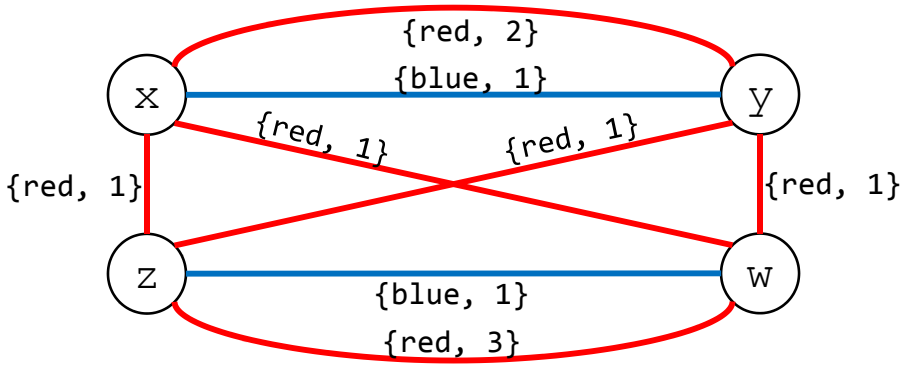

Figure 3.9. Dependency graph for $\mathsf{con}(x, y, a) \overset{1}{\approx} \mathsf{con}(z, b, w)$, $x \overset{2}{\approx} y$ and $z \overset{3}{\approx} w$.

and v, it introduces an equality of the form $u \approx \mathsf{con}(v, k)$. Similarly, when **F-Unify** detects a pair of variables, say u and v, it introduces an equality of the form $u \approx v$. Note that in the definition of either rule, u and v are atomic terms. When one of them is a string constant, the other term must be a string variable, and it will be colored as *black* after the application of either rule, as described in the construction of a dependency graph. In other word, a *black* node will not be used in later derivation. Therefore, we focus on the situation when both are string variables.

Definition 13 Let c be a configuration, u and v be two variables as declared in rules **F-Unify** and **F-Split**. Both the node u and the node v are white in the dependency graph of c. We call that u and v are *splittable* in configuration c. □

Lemma 10 Let c be an initial configuration in the acyclic form. If two variables u and v are *splittable*, then there exists a *compressed* path between the node u and the node v in the dependency graph of c.

Proof: Since it is an initial configuration, all nodes are white. If u and v are *splittable* variables with respect to either **F-Split** or **F-Unify** rule, there exist two terms s and t such that s and t are in the same equivalence class. Thus, the node for any variable in s has a compressed path to the node for any variable in t. Note that in our definition of a normal form, the normal form of an equivalence class that contains a string constant is always the (normalized) string constant.

Because both s and t have flat forms, u and v are variables and $\mathsf{F}\, s = (\boldsymbol{w}, u, \boldsymbol{u}_1)$ and $\mathsf{F}\, t = (\boldsymbol{w}, v, \boldsymbol{v}_1)$, there must exist a variable (say x) in s and a variable (say y) in t such that the normal form of $[x]$ contains u and the normal form of $[y]$ contains v. Thus, there is a compressed path p_2 between the node x and the node y. The way normalization rules are defined guarantees that there are a compressed path p_1 between the node x and the node u, and a compressed path p_3 between the node x and the node v.

Note that if the configuration is acyclic, a node x along with an edge (labeled with index i) connected to it defines the term that is on one side of the equality i and contains x. The compressed path from the node x to the node u describes how we built the normal form of $[x]$.

Now, we need to show that there is a compressed path between the node u and the node v. We prove that it is impossible to have edges labeled with the same index in p_1 and p_2. Proof is by contradiction. Assume that the index i is the shared index between p_1 and p_2. Then, there exist nodes u_1 and u_2 along the compressed path p_1, and nodes y_1 and y_2 along the compressed path p_2, such that the node x has

a compressed path to the node u_1, the node u_1 has a red edge with index i to the node u_2, the node u_2 has a compressed path to the node u, the node x has a compressed path to the node y_1, the node y_1 has a red edge with index i to the node y_2, and the node y_2 has a compressed path to the node y. Note that if $u_1 = x = y_1$, neither **F-Unify** nor **F-Split** is applicable. Indeed, the term defined by the node x and the edge labeled i connected to it along the path p_1, which illustrates the construction of the normal form of $[x]$, is the term x. The term defined by the node x and the edge labeled i connected to it along the path p_2 should be the same term x, as they come from the same equality indexed i. However, latter term defines the term s in the premise of **F-Unify** or **F-Split**. Thus, the terms x, s and t are in the same equivalence class. Then, since there exists a normal form of $[x]$, $\mathsf{F}\, s = \mathsf{F}\, t$, and neither **F-Unify** nor **F-Split** is applicable. So if there are several choices of index i, we can always pick the one for which the node u_1 has a compressed path to the node y_1.

Note that either $u_1 \neq x$ or $y_1 \neq x$. Without loss of generality, we can assume that $u_1 \neq x$, then there must be a blue edge with label i either between u_1 and y_1 or between u_1 and y_2. Thus, there is a cycle containing the nodes u_1 and y_1, but the configuration is acyclic in this lemma. We get the contradiction. Therefore, there is a compressed path between u and y. Similarly, we prove that there is a compressed path between x and v.

If there is a shared label between p_1 and p_3, there must be a cycle using a similar argument.

Therefore, there is a *compressed* path between u and v. ■

Lemma 11 Let c be a configuration in the acyclic form. If u and v are splittable, there is no direct blue edge between the node u and the node v in the dependency graph of c.

Proof: We prove the property by contradiction. Assume that there is a direct blue edge with the index j between the node u and the node v in the dependency graph of c. By Lemma 10, there is a *compressed* path p between u and v.

If the index j does not appear in the path p, by Definition 12, there is a cycle, which contradicts the assumption.

If the index j appears in the path p, we assume that in the path p there are two nodes, say z_1 and z_2, such that there is a compressed path between u and z_1 (without the index j), there is a compressed path between v and z_2 (without the index j), and there is a red edge between z_1 and z_2 with the index j. Thus, there is a direct edge (either blue or red) with the index j between u and z_1 if u is not equal to z_1; otherwise, there is a direct edge (either blue or red) with the index j between v and z_2. In either case, there is a cycle.

Therefore, there is no direct blue edge between u and v.

Lemma 12 (Acyclicity Preservation) Given an initial configuration in an acyclic form, after applying any rule in our calculus, the new configuration is still in an acyclic form.

Proof: By Lemma 9, all word equalities are linear, and thus, **F-Loop** is not applicable.

By Definition 12, the initial dependency graph does not contain a cycle. Since our calculus does not remove any word equalities from S and **F-Unify** and **F-Split** are the only rules that add new word equalities to S: it is enough to show that acyclicity is preserved after the application of **F-Unify** and **F-Split**. By Lemma 10 and Lemma 11, if two variables are splittable, there is a compressed path but no direct blue edge between the variables.

If a new configuration is the result of the application of **F-Split**, in the left branch, we add $u \approx \mathsf{con}(v, k)$ to S, as declared in the rule. In addition, we mark the node u as a black node. By Definition 12, any path containing the node u will not become a cycle in the dependency graph. Note that the fresh variable k, as declared in the rule **F-Split**, has only two edges: one red edge to the node u and one blue edge to the node v with a fresh index. If there is a new potential cycle, it is due to the new red edges and it must involve the node u, which is black, the acyclicity is preserved. Similarly, we have the preservation for the right branch.

If a new configuration is the result of the application of **F-Unify**, we mark the node u as a black node and add a new red edge between u and v with a fresh index. If there is a new potential cycle, it is due to this new red edge, thus, must contain the black node u.

Therefore, the acyclicity is preserved. ■

Definition 14 Let c be an acyclic configuration and x be a variable ($x \in \mathcal{V}(\mathsf{S})$), where S is the set of string constraints in c. The *candidate splitting set* for x with respect to c, denoted as Z_x^c, is a subset of $\mathcal{V}(\mathsf{S})$, such that a node y is in Z_x^c if

- the node y is a white node,
- there is a *compressed* path between the node x and the node y, and
- the node x and the node y has no direct *blue* edge between them. □

Note that the definition of Z_x^c is an *over-approximation* of the actual set of the variables that may be *splittable* with x.

Definition 15 Let c be an acyclic configuration and S be the set of string constraints in c. We define a *measure* of the configuration c, denoted as μ_c, as a descending ordered multi-set:

$$\mu_c := \{\ |Z_x^c|\ \mid\ \text{the node } x \text{ is a white node}, [x] \text{ contains only variables}\}, \tag{3.14}$$

where $|Z_x^c|$ is the cardinality of the set Z_x^c. □

Definition 16 We define the partial order on μ, denoted as $\prec_\mu$, as the standard definition of the multi-set order. □

Lemma 13 Let c be a configuration that is not *saturated* by either **F-Unify** or **F-Split**. Let c' be an immediate configuration after the application of either **F-Unify** or **F-Split** on c. Then, $\mu_{c'} \prec_\mu \mu_c$.

Proof: Assume c' is an immediate configuration after the application of **F-Unify** on c, and u, v are the atomic terms as declared in **F-Unify**. After asserting $u \approx v$ to S, either u or v is marked as black, and thus when building $\mu_{c'}$ either $|Z_u^{c'}|$ or $|Z_v^{c'}|$ should be removed from the measure. Assume that u is colored as black. For all node

x, if u is in Z_x^c, $Z_x^{c'}$ is equal to $Z_x^c - \{u\}$ by the construction of a dependency graph. Since no new node is added to the graph and $|Z_u^c|$ is removed from the multi-set $\mu_{c'}$, $\mu_{c'} \prec_\mu \mu_c$.

Assume c' is an immediate configuration after the application of **F-Split** on c, and u, v, k are the atomic terms as declared in **F-Split**. After asserting $u \approx \mathsf{con}(v, k)$, u is marked as black, and thus $|Z_u^{c'}|$ is not in the measure $\mu_{c'}$. Since the node k has only one red edge (with a fresh index) which is connected to the node u, any node which has a compressed path from the node u, also has a compressed path from the node k. No other node is connected to the node k. At the same time, since the node v is now directly connected to the node k with a blue edge, $Z_k^{c'}$ is equal to $Z_u^c - \{v\}$ and $|Z_k^{c'}| < |Z_u^c|$. Similarly, $Z_v^{c'}$ is equal to $Z_v^c - \{u\}$ and $|Z_v^{c'}| < |Z_v^c|$. For all other nodes x such that $u \in Z_x^c$, we have $Z_x^{c'}$ is equal to $Z_x^c - \{u\} + \{k\}$, and thus $|Z_x^c|$ is identical to $|Z_x^{c'}|$. The rest of the components in the measure remain unchanged. Thus, $\mu_{c'} \prec_\mu \mu_c$. ■

Strategy 3 The original proof strategy (see Section 3.1). In addition, we add the following requirements for derivation on constraints in the *acyclic* form:

- the derivation is under Assumption 2,
- we consider the derivation without **S-Cycle**,
- if a derivation tree is neither *closed* nor *saturated*, a rule is guaranteed to be applied,
- **Reset** only applied each time when S is changed, and

- the derivation stops immediately after a derivation tree is *saturated*.

□

Lemma 14 (Termination in Acyclicity) Let t be a derivation tree rooted by an initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, where S_0 is in the acyclic form. With Strategy 3, the derivation tree is finite.

Proof: The exchanged equalities between S and A are finite (by Lemma 5) between two resets.

By Lemmas 6, 7 and 8, all rules that do not modify S can only be applied for finitely many times between two resets.

By Strategy 3, we have no application of **D-Split** nor **S-Cycle**.

Because the configuration is acyclic, **F-Loop** will not be applied. Consequently, **R-Star** is not applicable.

By Lemma 13, the application of either **F-Unify** or **F-Split** decreases the measure. Thus, there is no infinite path in the derivation tree.

Therefore, the derivation tree is finite. ■

Theorem 5 (Decision Procedure on Acyclicity) *Let t be a derivation tree rooted by an initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, where S_0 is in the acyclic form. The derivation uses Strategy 3. If $S_0 \cup A_0$ is satisfiable, the derivation tree is saturated; otherwise, the derivation tree is closed.*

Proof: The exchanges between S and A are finite (by Lemma 5) and contain only

equalities (modeled by **A-Prop** and **S-Prop**). It follows the requirement of the Nelson-Oppen combination.

Because our calculus is refutation sound (by Theorem 1), it is solution sound (by Theorem 2) and it terminates on every acyclic configuration (by Theorem 14), it is a decision procedure for constraints in the acyclic form.

3.1.6 Implementation in DPLL(T)

Theory solvers based on the calculus we have described can be integrated into the DPLL(T) framework used by modern SMT solvers, which combines a SAT solver with multiple specialized *theory solvers* for conjunctions of constraints in a certain theory [14]. These SMT solvers maintain an evolving set F of quantifier-free clauses and a set M of literals representing a (partial) Boolean assignment for F. Periodically, a theory solver is asked whether M is satisfiable in its theory.

In terms of our calculus, we assume that the literals of an assignment M are partitioned into string constraints (corresponding to the set S), arithmetic constraints (the set A) and membership constraints (the set R). These sets are subsequently given to three independent solvers, which we will call the string solver, the arithmetic solver, and the membership solver, respectively. The rules **A-Prop** and **S-Prop** model the standard mechanism for Nelson-Oppen theory combination, where entailed equalities are communicated between these solvers [66]. The satisfiability check performed by the arithmetic solver is modeled by the rule **A-Conflict**. Note that there is no additional requirement on the arithmetic solver, and thus a standard DPLL(T) theory solver for linear integer arithmetic can be used. The behavior of the membership

solver is described by the rule **R-Star**. The remaining rules model the behavior of the string solver.

The case splitting done by the string solver (with rules **S-Split** and **L-Split**) is achieved by means of the *splitting on demand* paradigm [11], in which a solver may add theory lemmas to F consisting of clauses possibly with literals not occurring in M. The case splitting in rules **F-Split** and **D-Split** can be implemented by adding a lemma of the form $\psi \Rightarrow (l_1 \vee l_2)$ to F, where l_1 and l_2 are new literals. For instance, in the case of **F-Split**, we add the lemma $\psi \Rightarrow (u \approx \mathsf{con}(v, z) \vee v \approx \mathsf{con}(u, z))$, where ψ is a conjunction of literals in M entailing $s \approx t \wedge s \approx \mathsf{F}\, s \wedge t \approx \mathsf{F}\, t \wedge \mathsf{len}\, u \not\approx \mathsf{len}\, v$ in the overall theory.

The rules **Len**, **Len-Split**, and **Card** involve adding constraints to A. This is done by the string solver by adding lemmas to F containing arithmetic constraints. For instance, if $x \approx \mathsf{con}(y, z) \in \mathcal{K}(\mathsf{S})$, the solver may add a lemma of the form $\psi \Rightarrow \mathsf{len}\, x \approx \mathsf{len}\, y + \mathsf{len}\, z$ to F, where ψ is a conjunction of literals from M entailing $x \approx \mathsf{con}(y, z)$, after which the conclusion of this lemma is added to M (and hence to A).

In DPLL(T), when a theory solver determines that M is unsatisfiable (in the solver's theory) it generates a *conflict clause*, the negation of an unsatisfiable subset of M. The string solver maintains a compact representation of $\mathcal{K}(\mathsf{S})$ at all times.

To construct conflict clauses, the string solver also maintains an *explanation* $\psi_{s,t}$ for each equality $s \approx t$, when the equality is added to S by applying **S-Cycle**, **F-Unify** or standard congruence closure rules. The explanation $\psi_{s,t}$ is a conjunction

of string constraints in M such that $\psi_{s,t} \models_{\mathsf{SL}} s \approx t$. For **F-Unify**, the string solver maintains an explanation ψ for the flat form of each term $t \in \mathcal{D}(\mathsf{F})$ where $\psi \models_{\mathsf{SL}} t \approx \mathsf{con}(\mathsf{F}\, t)$. When a configuration is determined to be unsatisfiable by **S-Conflict**, that is, when $s \approx t, s \not\approx t \in \mathcal{K}(\mathsf{S})$ for some s, t, it replaces the occurrence of $s \approx t$ with its corresponding explanation ψ, and then replaces the equalities in ψ with their corresponding explanation, and so on, until ψ contains only equalities from M. Then it reports as a conflict clause (the clause form of) $\psi \Rightarrow s \approx t$.

All other rules (such as those that modify N, F and B) model the internal behavior of the string solver.

Now, we describe how to integrate the *fair* strategy into the DPLL(T) framework. The idea is similar to *Finite Model Finding* strategy discussed in [77]. We use J to denote the term for the sum of lengths of all input variables, i.e., $\Sigma_{x \in \mathcal{V}(S_0)} \mathsf{len}(x)$, where S_0 is the set of input constraints. Facilitated by *split-on-demand* in DPLL(T) [11], we mark a new literal $J \leq 0$ as the *first decision literal* [10] in the DPLL(X) engine, initially.

Each time, when the string solver gets a set of literals from the DPLL(X) engine, it first checks the polarity of the literal $J \leq n$ (where n is a non-negative integer constant). If the polarity is positive, the string engine continues with the normal procedure; otherwise, it tries to mark a new literal $J \leq n + 1$ as the *first decision literal* in the DPLL(X) engine.

[10] see definition in Section 1.2

3.2 Constant Splitting Refinement

In Figure 3.5, we described the **F-Split** rule. The **F-Split** generally explains how our calculus splits an atomic term into two: it branches the search space into two, and on each branch one atomic term is transformed into a concatenation containing a fresh variable.

In **F-Split**, both u and v are atomic terms, i.e., either a constant or a variable. When one is a constant and the other is a variable, we can apply a heuristic that performs extensive propagation instead of splitting. First, we introduce some definitions.

Definition 17 Let l_a and l_b be two string constants. We say l_b defines a *cut string* over l_a, denoted as l_c, if $l_a = \mathsf{con}(l_{a'}, l_c)$ and, either (i) $l_b = \mathsf{con}(l_c, l_{b'})$ or (ii) $l_c = \mathsf{con}(l_b, l_{c'})$, where $l_c, l_{a'}, l_{b'}, l_{c'}$ are string constants, not necessarily non-empty. □

Case (i) in the definition represents the case when l_a and l_b share common suffix and prefix respectively, i.e., l_c is a suffix of l_a and a prefix of l_b at the same time.

Case (ii) represents the case when l_b is a substring of l_a. In this case, $l_{c'}$ is a suffix of l_a that follows an occurrence of l_b in l_a, i.e., $l_a = \mathsf{con}(l'_a, l_b, l_{c'})$ and $l_c = \mathsf{con}(l_b, l_{c'})$.

Definition 18 For an arbitrary pair of string constants l_a and l_b, in general l_b may define a set of the *cut strings* over l_a. Among the elements of this set, we define the *maximal cut string* (produced by l_b over l_a) to be the one with the *longest* length,

$$\textbf{H-C-Split}\quad \frac{\begin{array}{c} s \approx t \in \mathcal{K}(\mathsf{S}) \quad \mathsf{F}\, s = (\boldsymbol{w}, l_a, \boldsymbol{u}) \quad \mathsf{F}\, t = (\boldsymbol{w}, x, l_b, \boldsymbol{v}) \\ x \notin \mathcal{V}(\boldsymbol{u}) \quad x \not\approx \epsilon \quad l_a \approx \mathsf{con}(l_{a'}, l_{mc}) \end{array}}{\mathsf{S} := \mathsf{S}, x \approx \mathsf{con}(l_{a'}, k)}$$

Figure 3.10. Constant splitting rule, where s, t are terms of String, x is a string variable, $\boldsymbol{w}, \boldsymbol{u}, \boldsymbol{v}$ are vectors of String terms, and l_{mc} is the *maximal cut string* produced by l_b over l_a.

denoted as l_{mc}. □

We use maximal cut strings to prompt propagation in certain cases. This idea is presented as a constant splitting rule, given in Figure 3.10.

Correctness

This rule can be considered as a special case of **F-Split** where one atomic term is a constant and the other atomic term is a variable followed by a constant.

Generally speaking, to prove the soundness of a rule, we need to show that any model satisfying the premises, satisfies the conclusion. In another word, no model is missing during a derivation.

With respect to the rule **F-Split**, noticing that $l_{a'}$ is a prefix of l_a and the conclusion introduces a new constraint for x, we want to show that for every model of x, it must have the prefix $l_{a'}$.

Theorem 6 (Correctness for Constant Splitting) *The* **H-C-Split** *rule is sound.*

Proof: We prove this property by contradiction. Assume x has a potential model l' whose prefix is not $l_{a'}$, where l_{mc} is the maximal cut string produced by l_b over l_a and $l_a = \mathsf{con}(l_{a'}, l_{mc})$.

If $\mathsf{len}(l') \geq \mathsf{len}(l_{a'})$, since $\mathsf{con}(\boldsymbol{w}, l_a, \boldsymbol{u}) = \mathsf{con}(\boldsymbol{w}, x, l_b, \boldsymbol{v})$ and $l_a = \mathsf{con}(l_{a'}, l_{mc})$, $l_{a'}$ has to be the prefix of l', and thus $l' = \mathsf{con}(l_{a'}, l'')$ which contradicts the assumption.

If $\mathsf{len}(l') < \mathsf{len}(l_{a'})$, because $l_a = \mathsf{con}(l_{a'}, l_{mc})$, we have $l_{a'} = \mathsf{con}(l', l'')$, where l'' is not empty. By applying the substitution $x \mapsto l'$ and dropping the common prefix, we have $\mathsf{con}(l'', l_{mc}, \boldsymbol{u}) \approx \mathsf{con}(l_b, \boldsymbol{v})$. Notice three facts: l_{mc} is the maximal cut string produced by l_b over l_a l'' is non-empty and l_b is non-empty. If $\mathsf{con}(l'', l_{mc}, \boldsymbol{u}) \approx \mathsf{con}(l_b, \boldsymbol{v})$ is satisfiable and l' is a model of x, l'' and l_b must share a common prefix. In another word, $\mathsf{con}(l'', l_{mc})$ is a cut string produced by l_b over l_a. Clearly, $\mathsf{con}(l'', l_{mc})$ is longer than l_{mc}, which contradicts the assumption that l_{mc} is the maximal cut string.

Therefore, the rule is sound. ■

3.3 String Manipulating Function Extension

One goal of this project is to devise a language-independent string solver to solve string constraints generated by tools for security analysis. This goal requires us to consider the following two aspects:

- *Expressiveness*: wide coverage of string functions in programming languages, and
- *Accuracy*: precise modeling (similar to the proposed evaluation by [50]), i.e., the cost of conversion from a string function (in a programming language) to the one in our language should be as low as possible, and string functions should as less approximation as possible.

To achieve the expressiveness, we adopt the core SMT-LIB language [13] (which defines a language for automated reasoning) and extend the language based on a proposal for the theory of sequences [16]. Note that the theory of strings is mainly a special case of the theory of sequences, where a string is a sequence of characters. However, the theory of sequences does not cover all signatures from the theory of strings. For example, type conversion is one of the most frequently used functions in the security analysis; whereas in the sequence theory, the underlying character sort does not require to have an ordering and thus there is no type conversion definition. To compensate the signature shortage proposed for the theory of sequences (but useful for strings), we have consulted [42, 55, 33] to extend our language.

To achieve the accuracy, our deduction does not rely on any kind of approximation. Note that solving constraints containing a common string manipulating function is more like model checking (e.g., inductive checking may help our reasoning), and certain abstraction may be involved for performance, e.g., [96]. Generally speaking, if an over-approximation (of a string function behavior) occurs in a constraint solver, the model may not satisfy the constraints, and thus the solver is not solution complete; whereas if an under-approximation occurs, the solver may report `unsat` even thought there is a satisfying model, and this leads to unsoundness. Since using approximations conflicts with the purpose of an automated reasoning over strings, our rules strictly follow the theory of strings.

The list of extended string manipulating functions, together with informal specification, are given in Table 3.1. Our rules and proofs rely on these definitions.

Function	Specification
$\mathsf{char\_at}(s, i) \approx t$	t is the character of s at the position i, if i is non-negative term and smaller than the length of s; otherwise, it is undefined.
$\mathsf{substr}(s, i, j) \approx t$	t is the substring of s of length j starting position i, if i, j are non-negative terms and the sum of i and j is smaller than the length of s; otherwise, it is undefined.
$\mathsf{contains}(s, t) \approx b$	b is true iff t is a substring of s.
$\mathsf{index\_of}(s, t, i) \approx j$	j is the position of the first occurrence of t in s after the index i; $j \approx -1$ if t does not occur in s. t is non-empty, the value of i is a non-negative.
$\mathsf{replace}(s, t_1, t_2) \approx t$	t is a string by replacing the first occurrence of t_1 in s with t_2 if s contains t_1; t is s, otherwise.
$\mathsf{prefix\_of}(s, t) \approx b$	b is true iff s is a prefix of t.
$\mathsf{suffix\_of}(s, t) \approx b$	b is true iff s is a suffix of t.
$\mathsf{str\_to\_int}(s) \approx i$	i is the corresponding natural number of s in decimal notation if s contains only digits; $i \approx -1$, otherwise.
$\mathsf{int\_to\_str}(i) \approx t$	t is the corresponding string if the value of i is non-negative; t is the empty string, otherwise.

Table 3.1. Definitions for string manipulating functions, where s, t, t_1, t_2 are terms of String, b is a term of Bool, and i, j are terms of Int. The index is starting from 0. The function $\mathsf{char\_at}$ and substr are partial functions. t_1 is non-empty.

In addition, we require the alphabet contains all digital characters ("0", ..., "9").

To handle these functions, we extend our calculus with a set of new rules. To support those rules, we extend our configuration with two new components G and Q. The set G is a set of ground formulas over the extended signature. The set Q is a set of quantified formulas over the extended signature. Initially, all input constraints are distributed into these two sets according to their structure.

$$\textbf{E-CharAt}\quad \frac{\mathsf{char\_at}(s,i) \approx t \in \mathsf{S}}{\mathsf{S} := \mathsf{S}, \mathsf{substr}(s,i,1) \approx t}$$

$$\textbf{E-Substr}\quad \frac{\mathsf{substr}(s,i,j) \approx t \in \mathsf{S}}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(k_1,t,k_2) \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(k_1) \approx i, \mathsf{len}(t) \approx j, \\ 0 \le i, 0 < j, i+j < \mathsf{len}(s) \quad \| \\ \mathsf{A} := \mathsf{A}, 0 > i \quad \| \quad \mathsf{A} := \mathsf{A}, 0 \ge j \quad \| \quad \mathsf{A} := \mathsf{A}, i+j \ge \mathsf{len}(s)\end{array}}$$

Figure 3.11. Rules for handling the substring functions, where s, t are terms of String, i, j are terms of Int, and k_i's are fresh terms of String.

3.3.1 Extended Calculus for String Manipulating Functions

In Figure 3.11, we describe a set of rules that handles char_at and substr functions. The semantics follow the standard definitions of these functions in programming languages, as in Table 3.1. The index always starts with 0.

E-CharAt reduces the function char_at to substr. **E-Substr** reduces the function substr to concatenation. Notice that the function substr is a partial function in SMT-LIB. If an input to substr is not in the domain, the behavior is undefined.

In Figure 3.12, we describe a set of rules that handles string function contains. **E-Contains-P** deals with positive contains predicates by converting them into concatenations. **E-Contains-N** converts negative contains predicates into quantified formulas. The philosophy behind this rule is that: when a string s is not a substring of a string t, then for each substring of t with the same length as s, there exists at least one position where s has a different character of the substring of t.

Notice that in case of **E-Contains-N** rule, the resulting quantified formula is of a special type: quantified variables are bounded by a particular range (although the ranges may be symbolic). This allows an SMT solver to utilize a corresponding

strategy, which we call *bounded integer quantification*. Bounded integer quantification is a special extension of the fair strategy. It is refutation sound and model complete [76]. Since the modeling procedure is precise without any approximation, these string functions are also refutation sound and model complete, i.e., if the constraints are satisfiable, the produce can find a model if fairness is applied. By fairness, we mean the procedure will not stick to one certain branch without exploring other branches.

Rules in Figure 3.11 and Figure 3.12 form the core rules for handling extended string functions. Other string functions are reduced to substr, contains, concatenation and bounded integer quantification in this calculus. A general function reduction flow is shown in Figure 3.13. Notice that solving word equations with length constraints is essential for these string manipulating functions. Also, substr is a core function in this set of rules because most of these functions can be reduced to substr. Although we reduce substr into word equations, it might be more effective if we can treat it directly as a primitive data structure in the theory of strings. contains is the only function that is reduced to quantified formulas, although both index_of and replace depend on this reduction.

In Figure 3.14, we describe a set of rules that convert additional string manipulating functions to quantifier-free word equations and the function contains. The conversion follows the semantics of these functions. Notice that if the negative contains functions together with length constraints are decidable, these functions are decidable too, although it has been proved that replace is undecidable [89].

E-Contains-P
$$\frac{\mathsf{contains}(s,t) \in \mathsf{G}}{\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(k_1, t, k_2)}$$

E-Contains-N
$$\frac{\neg\mathsf{contains}(s,t) \in \mathsf{G}}{\begin{array}{l}\mathsf{Q} := \mathsf{Q}, \forall i : \mathsf{Int}.\exists j : \mathsf{Int}.\ \mathsf{implies}(0 \leq i \leq \mathsf{len}(s) - \mathsf{len}(t), \\ \quad 0 \leq j \leq \mathsf{len}(t) \wedge \mathsf{char\_at}(s, i+j) \not\approx \mathsf{char\_at}(t, j))\end{array}}$$

Figure 3.12. Rules for handling the contains function, where s, t are terms of String, and k_i's are fresh terms of String.

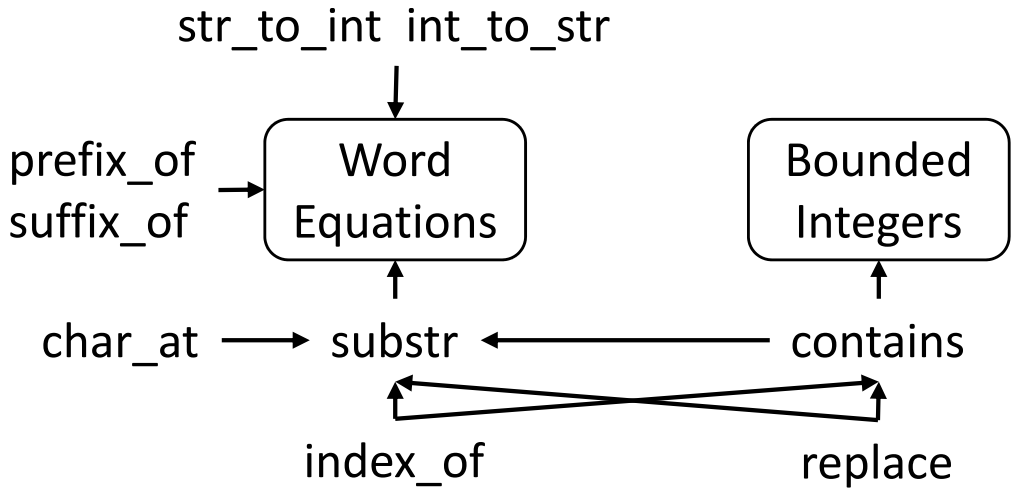

Figure 3.13. Reduction flows for string manipulating functions.

Figures 3.15 and 3.16 present the rules for handling type conversion functions.

The function `is_digit` is a shorthand for a disjunction of equalities for testing whether a string term is a digital character, i.e.,

$$\texttt{is\_digit}(s) \quad \equiv \quad s \approx \text{“0”} \vee s \approx \text{“1”} \vee \cdots \vee s \approx \text{“9”}. \tag{3.15}$$

The function `char_to_int` is another shorthand for the ite term that converts a single digital character s to corresponding digit, i.e.,

$$\texttt{char\_to\_int}(s) \quad \equiv \quad \mathsf{ite}(s \approx \text{“0”}, 0, \mathsf{ite}(s \approx \text{“1”}, 1, \cdots, 9)\cdots). \tag{3.16}$$

E-IndexOf
$$\frac{\mathsf{index\_of}(s,p,i) \approx j \in \mathsf{S}}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(k_1,k_2,p,k_3) \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(k_1) \approx i, j \approx i + \mathsf{len}(k_2) \\ \mathsf{G} := \mathsf{G}, \neg\mathsf{contains}(\mathsf{con}(k_2, \mathsf{substr}(p,0,\mathsf{len}(p)-1)),p) \\ \| \\ \mathsf{A} := \mathsf{A}, j \approx -1 \quad \mathsf{G} := \mathsf{G}, \neg\mathsf{contains}(\mathsf{substr}(s,i),p)\end{array}}$$

E-Replace
$$\frac{\mathsf{replace}(s,p,q) \approx t \in \mathsf{S}}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(k_1,p,k_2), t \approx \mathsf{con}(k_1,q,k_2) \\ \mathsf{G} := \mathsf{G}, \neg\mathsf{contains}(\mathsf{con}(k_1, \mathsf{substr}(p,0,\mathsf{len}(p)-1)),p), \mathsf{contains}(s,p) \\ \| \\ \mathsf{S} := \mathsf{S}, s \approx t \quad \mathsf{G} := \mathsf{G}, \neg\mathsf{contains}(s,p)\end{array}}$$

E-PrefixOf
$$\frac{\mathsf{prefix\_of}(s,t) \in \mathsf{G}}{\mathsf{S} := \mathsf{S}, t \approx \mathsf{con}(s,k)}$$

E-SuffixOf
$$\frac{\mathsf{suffix\_of}(s,t) \in \mathsf{G}}{\mathsf{S} := \mathsf{S}, t \approx \mathsf{con}(k,s)}$$

Figure 3.14. Rules for handling additional string manipulating functions, where s, t, p, q are terms of String, k_i's are fresh terms of String, and i, j are terms of Int.

Type conversion rules convey a similar idea as the *fair* strategy. Thus, application of these rules requires a fair strategy to be solution complete.

3.3.2 Handling the new components G and Q

In Figure 3.17, **G-Conj**, **G-Disj** and **G-Ite** handle various logical structures in a formula. They are designed to dissemble a formula into literals.

G-Str distributes (dis-)equalities to S. Notice that the congruence closure is constructed only for S. **G-Lia** and **G-Rex** distribute linear arithmetic formulas and membership constraints to A and R, respectively.

These rules are applied at the very beginning and right after the rule **Reset**. Notice that string predicates remain in G, e.g., contains, prefix_of and suffix_of. They remain in G only. At a highly abstract level, these G-rules mimic how the DPLL(X)

$$\textbf{E-Int2Str-1}\quad \frac{\mathsf{int\_to\_str}(n) \approx s \in \mathsf{S} \quad \mathsf{A} \models n < 0}{\mathsf{S} := \mathsf{S}, s \approx \epsilon}$$

$$\textbf{E-Int2Str-2}\quad \frac{\mathsf{int\_to\_str}(n) \approx s \in \mathsf{S} \quad \mathsf{A} \models 0 \leq n < 10}{\mathsf{S} := \mathsf{S}, n \approx \mathtt{char\_to\_int}(s), \mathsf{len}(s) \approx 1}$$

$$\textbf{E-Int2Str-3}\quad \frac{\mathsf{int\_to\_str}(n) \approx s \in \mathsf{S} \quad \mathsf{A} \models n \geq 10}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(s_1, s_2), \mathsf{len}(s_1) \geq 1, \mathsf{len}(s_2) = 1 \\ \mathsf{A} := \mathsf{A}, n_1 > 0, n_2 \geq 0, n \approx n_1 \times 10 + n_2, \\ \mathsf{int\_to\_str}(n_1) \approx s_1, \mathsf{int\_to\_str}(n_2) \approx s_2\end{array}}$$

Figure 3.15. Rules for handling the int_to_str type conversion function, where s is a term of String, s_1, s_2 are fresh terms of String, n is a term of Int, and n_1, n_2 are fresh terms of Int.

engine distributes partial models to its plug-in theory engines.

Note that initially Q is empty and only the rule **E-Contains-N** introduces new formulas to Q. The formula introduced by the rule are of the special form. Since this thesis does not focus on how to solve quantified formulas, we use a fairly simple rule **Q-Inst** in Figure 3.18 to mimic the approach for handling quantified formulas. The actual approach is more sophisticated and discussed in [76].

3.3.3 Correctness

In this subsection, we prove the correctness of the rules modeling string manipulating functions.

Lemma 15 **E-Substr** is sound.

Proof: Assume that $s \approx c_0 \cdots c_i \cdots c_{i+j} \cdots c_{n-1}$. Notice that $0 \leq i, i+j \leq n-1 < n = \mathsf{len}(s)$. By definition of substr, $t \approx c_i \cdots c_{i+j}$, which is a substring of s. In the conclusion of **E-Substr**, we have $s \approx \mathsf{con}(k_1, t, k_2), \mathsf{len}(k_1) \approx i, \mathsf{len}(t) \approx j$. So,

E-Str2Int-1 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S}}{\mathsf{A} := \mathsf{A}, n \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, n \geq 0}$$

E-Str2Int-2 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{S} \models s \approx \epsilon}{\mathsf{A} := \mathsf{A}, n \approx -1}$$

E-Str2Int-3 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{A} \models n \geq 0}{\mathsf{S} := \mathsf{S}, \mathsf{len}(s) \approx 1 \quad \| \quad \mathsf{S} := \mathsf{S}, \mathsf{len}(s) \geq 1}$$

E-Str2Int-4 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{S} \models \mathsf{len}(s) \approx 1, n \geq 0}{\mathsf{A} := \mathsf{A}, n \approx \mathtt{char\_to\_int}(s) \quad \mathsf{G} := \mathsf{G}, \mathtt{is\_digit}(s)}$$

E-Str2Int-5 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{S} \models \mathsf{len}(s) \approx 1, n \approx -1}{\mathsf{G} := \mathsf{G}, \neg \mathtt{is\_digit}(s)}$$

E-Str2Int-6 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{A} \models \mathsf{len}(s) > 1, n \geq 0}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(s_1, s_2), \mathsf{str\_to\_int}(s_1) \approx n_1, \mathsf{str\_to\_int}(s_2) \approx n_2 \\ \mathsf{A} := \mathsf{A}, \mathsf{len}(s_2) \approx 1, n \approx n_1 \times 10 + n_2, n_1 > 0, 0 \leq n_2 < 10\end{array}}$$

E-Str2Int-7 $$\frac{\mathsf{str\_to\_int}(s) \approx n \in \mathsf{S} \quad \mathsf{A} \models \mathsf{len}(s) > 1, n \approx -1}{\begin{array}{c}\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(s_1, s_2), \mathsf{str\_to\_int}(s_1) \approx n_1, \mathsf{str\_to\_int}(s_2) \approx n_2 \\ (\mathsf{A} := \mathsf{A}, \mathsf{len}(s_2) \approx 1, n_1 \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(s_2) \approx 1, n_2 \approx -1)\end{array}}$$

Figure 3.16. Rules for handling the $\mathsf{str\_to\_int}$ type conversion function, where s is a term of String, s_1, s_2 are fresh terms of String, n is a term of Int, and n_1, n_2 are fresh terms of Int.

$k_1 \approx c_0 \cdots c_{i-1}$, $t \approx c_i \cdots c_{i+j}$ and thus $t \approx c_i \cdots c_{i+j}$. ■

Lemma 16 **E-CharAt** is sound.

Proof: Assume that $s \approx c_0 \cdots c_i \cdots c_{n-1}$. Notice that $0 \leq i \leq n-1 < n = \mathsf{len}(s)$.

By definition of $\mathsf{char\_at}$, $ift \approx c_i$, then it is $\mathsf{substr}(s, i, 1)$. ■

Lemma 17 **E-Contains-P** and **E-Contains-N** are sound.

Proof: If $\mathsf{contains}(s, t)$ is true, t is a substring of s. In another word, $\exists k_1, k_2.s \approx \mathsf{con}(k_1, t, k_2)$. Thus, **E-Contains-P** is sound.

$$\textbf{G-Conj} \quad \frac{\mathsf{G} = \mathsf{G}', \phi_1 \wedge \phi_2}{\mathsf{G} := \mathsf{G}', \phi_1, \phi_2}$$

$$\textbf{G-Disj} \quad \frac{\mathsf{G} = \mathsf{G}', \phi_1 \vee \phi_2}{\mathsf{G} := \mathsf{G}', \phi_1 \quad \| \quad \mathsf{G} := \mathsf{G}', \phi_2}$$

$$\textbf{G-Ite} \quad \frac{\mathsf{G} = \mathsf{G}', \mathsf{ite}(\phi_1, \phi_2, \phi_3)}{\mathsf{G} := \mathsf{G}', \phi_1, \phi_2 \quad \| \quad \mathsf{G} := \mathsf{G}', \neg\phi_1, \phi_3}$$

$$\textbf{G-Str} \quad \frac{\mathsf{G} = \mathsf{G}', (\neg)s \approx t}{\mathsf{G} := \mathsf{G}', \mathsf{S} := \mathsf{S}, (\neg)s \approx t}$$

$$\textbf{G-Lia} \quad \frac{\mathsf{G} = \mathsf{G}', a}{\mathsf{G} := \mathsf{G}', \mathsf{A} := \mathsf{A}, a}$$

$$\textbf{G-Rex} \quad \frac{\mathsf{G} = \mathsf{G}', (\neg)\mathsf{in}(s, R)}{\mathsf{G} := \mathsf{G}', \mathsf{R} := \mathsf{R}, (\neg)\mathsf{in}(s, R)}$$

Figure 3.17. Rules for handling the quantifier-free formula set G, where s, t are string terms and a is an arithmetic literal

$$\textbf{Q-Inst} \quad \frac{\forall i : \mathsf{Int}.\exists j : \mathsf{Int}.\ \mathsf{implies}(A[i], B[i, j]) \in \mathsf{Q} \quad n \text{ is a ground term of type } \mathsf{Int}}{\mathsf{G} := \mathsf{G}, A[n], B[n, m] \quad \| \quad \mathsf{G} := \mathsf{G}, \neg A[n]}$$

Figure 3.18. Rules for handling the quantified formula set Q, where m is a fresh integer constant.

If $\mathsf{contains}(s, t)$ is false, t is not a substring of s. In other words, $\forall k.\mathsf{contains}(s, k)$ implies $k \not\approx t$. If two strings have different length, they are definitely different. This formula can be further optimized by only checking all these substrings of s have the same length as t. If two strings of the same length are different, then there must be

a position at which the two strings have distinct characters. Therefore, we have:

$$\begin{aligned}&\mathsf{contains}(s,t) \iff \\ &\qquad (\forall i. \quad 0 \le i \le \mathsf{len}(s) - \mathsf{len}(t) \implies \\ &\qquad\qquad \exists j. 0 \le j \le \mathsf{len}(t) \wedge \mathsf{char\_at}(s, i+j) \not\approx \mathsf{char\_at}(t, j))\end{aligned} \tag{3.17}$$

■

Lemma 18 **E-IndexOf** is sound.

Proof: Assume that $s \approx c_0 \cdots c_{i-1} c_i \cdot \cdots \cdot c_{n-1}$. Let p be a string that does not occur in $c_i \cdot \cdots \cdot c_{n-1}$, by definition, the function returns -1. This is reflected in the right branch of the rule.

Assume that $p \approx c_j \cdots c_{j+m-1}$ and $s \approx c_0 \cdots c_{i-1} c_i \cdots c_j \cdots c_{j+m-1} \cdots c_{n-1}$. By definition, the function returns j, which is the first occurrence after the position i. Thus, s is in the form of $\mathsf{con}(k_1, k_2, p, k_3)$, where $\mathsf{len}(k_1) \approx i$, $j \approx i + \mathsf{len}(k_2)$ and $\neg\mathsf{contains}(\mathsf{con}(k_2, \mathsf{substr}(p, 0, \mathsf{len}(p) - 1)), p)$ (first occurrence by definition). ■

Lemma 19 **E-Replace** is sound.

Proof: By definition, if s does not contain p, the function returns p, shown in the right branch of the rule.

If s contains p, s is in the form of $\mathsf{con}(k_1, p, k_2)$. By definition, p has to be the first occurrence, so this is equivalent to $\neg\mathsf{contains}(\mathsf{con}(k_1, \mathsf{substr}(p, 0, \mathsf{len}(p) - 1)), p)$ and $t \approx \mathsf{con}(k_1, q, k_2)$. ■

Lemma 20 **E-PrefixOf** is sound.

Proof: If s is a prefix of t, t must be able to be partitioned into two substrings, where the first part is s. ■

Lemma 21 **E-SuffixOf** is sound.

Proof: If s is a suffix of t, we must be able to partition t into two substrings, where the second part is s. ■

Lemma 22 Rules for handling int_to_str are sound.

Proof: If s is the result of int_to_str(n), we splits n by its value, namely:

- if $n < 0$, by definition, s is the empty string;
- if $0 \leq n < 10$, by definition, s is a single character corresponding to n;
- if $10 \leq n$, the resulting string s must be at least of length 2. We divide n into two numbers, n_1 and n_2, such that $n = n_1 \times 10 + n_2$. Also, we divide s into two substrings s_1 and s_2, where the suffix s_2 is of length 1. We know that s_1 is the result value of int_to_str(n_1) and s_2 is the result value of int_to_str(n_2).

■

Lemma 23 Rules for handling str_to_int in Figure 3.16 are sound.

Proof: The rules for handling str_to_int adopt a similar idea as the *fair* strategy. We require the procedure to explore the left branch first, as usual; if it fails to find a model, then proceeds to the right branch.

By definition, the str_to_int function returns -1 if s is invalid (i.e., either s is the empty string or s contains non-digit characters); otherwise, it returns a non-negative number. Thus, **E-Str2Int-1** is sound. Another base case is when s is an empty string, the function returns -1. **E-Str2Int-2** is sound.

If s is valid, we try to get the length of s by **E-Str2Int-3**. Notice that we always try the left branch first by assumption. After applying this rule, we know that the length of s is at least one.

If s is a valid input and its length is 1, then s must be a digit character, and we can convert s to its corresponding number. If s is invalid and its length is 1, s must not be a digit character. This concludes the correctness of **E-Str2Int-4** and **E-Str2Int-5**.

If the length of s is greater than 1, we try to break the string into two substrings, where the second one is of length 1. If s is invalid, one of the substrings must be invalid (by **E-Str2Int-7**); otherwise, we know that the converted number should satisfy: $n = n_1 \times 10 + n_2$ and $0 \leq n_2 < 10$ (by **E-Str2Int-6**).

It concludes the soundness proof for the str_to_int rules. ■

3.4 Experimental Results

We have implemented a theory solver based on the calculus and the proof procedure described in the previous section within the latest version of our SMT solver CVC4. The string alphabet $\mathcal{A}$ for this implementation is the set of all 256 extended ASCII characters. To evaluate our solver we did an experimental comparison with two of the string solvers mentioned in Chapter 5: Z3-STR (version 20140120) and

KALUZA (latest version from its website). These solvers, which have been widely used in security analysis, were chosen because they are publicly available and have an input language that largely intersects with that of our solver. All results in this section were collected on a 2.53 GHz Intel Xeon E5540 with 8 MB cache and 12 GB main memory.[11]

We do not collect enough benchmarks over our extended language from the industry (by the time of this thesis submission) to produce substantial results, and thus the benchmarks we used in the experiments focus on word equations and length constraints. The original benchmarks are from the KALUZA project [12]. We translated these into SMT-LIB format with CVC4-style string constraints. Finally, these SMT-LIB versions were turned into Z3-str's format (same as SMT-LIB but the String constraints are slightly different, see below). Each version contains 47,284 benchmarks.

Modulo superficial differences in the concrete input syntax, all three tools accept as input a set of T_{SL} constraints and report on its satisfiability with a sat, unsat or unknown answer. In the first case, CVC4 and Z3-STR can also provide a *solution*, i.e., a satisfying assignment for the variables in the input set. KALUZA can do that for at most one *query variable* which must be specified before-hand in the input file.

An initial series of regression tests on all three tools revealed several usability

[11] Detailed results and binaries can be found at `http://cvc4.cs.nyu.edu/papers/CAV2014-strings/`.

[12] Available at: `http://webblaze.cs.berkeley.edu/2010/kaluza/`.

and correctness issues with KALUZA and a few with Z3-STR. In KALUZA, they were caused by bugs in its top level script which communicates with different tools (e.g., the solvers Yices and Hampi) via the file system. They range from failure to clean up temporary files to an incorrect use of the Unix grep tool to extract information from the output of those tools. Since KALUZA is not in active development anymore, we made an earnest, best effort attempt to fix these bugs ourselves. However, there seem to be more serious flaws in KALUZA's interface or algorithm. Specifically, often KALUZA incorrectly reports unsat for problems that are satisfiable only if some of their input variables are assigned the empty string. Moreover, in several cases, KALUZA's sat/unsat answer for the same input problem changes depending on the query variable chosen. Because of this arbitrariness, in our experiments we removed all query variables in KALUZA's input.

We found that in several cases Z3-STR returns *spurious solutions*, assignments to the input variables that do not in fact satisfy the input problem. Also, it classifies some satisfiable problems as unsat. Prompted by our inquiries, the Z3-STR developers have produced a new version of Z3-STR that fixes the spurious solutions problem. Unfortunately, that version was not ready in time for us to redo the experiments. As for Z3-STR's unsoundness, it looks like it is caused by an internal restriction that, for efficiency but without loss of generality, limits the possible values of "free" string variables to a fixed finite set of string constants. The authors define a variable as free in an input problem if its values are completely unconstrained by the problem. For instance, in the constraint set $\{x \approx \mathsf{con}(y, z)\}$ variables y and z would be free

	CVC4	Z3-str		Kaluza		Kaluza-orig	
Result		×	✓	×	✓	×	✓
unsat	11,625	317	11,769	7,154	13,435	27,450	805
sat	33,271	1,583	31,372	n/a	25,468	n/a	3
unknown	0		0		3		0
timeout	2,388		2,123		84		84
error	0		120		1,140		18,942

Table 3.2. Comparative results with CVC4, Z3-STR and KALUZA, over KUDZU benchmarks.

according to this definition, while x would not. It appears that the criterion used by Z3-STR to recognize free variables sometimes misclassifies a variable as free when in fact it is not, causing the system to miss solutions that are outside the finite domain imposed on free variables.

In contrast, on our full set of benchmarks, we did not find any evidence of erroneous behavior in CVC4 when compared with the other two solvers. Every solution produced by CVC4 was *confirmed* by both CVC4 and Z3-STR by adding the solution as a set of constraints to the input problem and checking that the strengthened problem was satisfiable. Furthermore, no unsat answers from CVC4 were contradicted by a confirmed solution from Z3-STR.

For our comparative evaluation we selected 47,284 benchmark problems from a set of about 50K benchmarks generated by Kudzu, a symbolic execution framework for Javascript, and available on the KALUZA website [82]. The discarded problems either had syntax errors or included a macro function (CapturedBrack) whose meaning is not fully documented. We translated those benchmarks into CVC4's extension of

the SMT-LIB 2 format to the language of T_{SL}[13] and into the Z3-STR format. Some benchmarks contain regular membership constraints $\mathsf{in}(s, r)$, which Z3-STR does not support. However, in all of these constraints the regular language denoted by r is finite and small, so we were able to translate them into equivalent string constraints.

We ran CVC4, Z3-STR and two versions of KALUZA, the original one and the one with our debugged main script, on each benchmark with a 20-second CPU time limit. The results are summarized in Table 3.2. There, the column Kaluza-orig refers to the original version of KALUZA while the error line counts the total number of runtime errors. The results for Z3-STR and the two versions of KALUZA are separated in two columns: the × column contains the number of provably incorrect answers while the ✓ column contains the rest. By *provably incorrect* here we mean an unsat answer for a problem that has a verified solution or a sat answer but with a spurious solution. Note that the figures for the two versions of KALUZA are unfairly skewed in their favor because neither version returns solutions, which means that their sat answers are unverifiable unless one of the other solvers produces a solution for the same problem. For a more detailed discussion, we look at the benchmark problem set broken down by the CVC4 results. For brevity we discuss only our amended version of KALUZA below.

None of the 11,625 unsat answers provided by CVC4 were provably incorrect. Z3-STR also answered sat on 11,568 of them and returned an error for the remaining 57; KALUZA agreed on 11,394 and returned an error for the rest. All of CVC4's 33,271

[13] CVC4's string extension is documented at `http://cvc4.cs.nyu.edu/wiki/Strings`.

sat answers were corroborated by a confirmed solution. Z3-STR agreed on 31,616 of those problems although it returned a spurious solution for 244 of them. Also, it incorrectly found 317 problems unsatisfiable and produced an error on 29 problems, timing out on the remaining 1,304. KALUZA agreed on 25,468 problems (unverifiable because of the absence of solutions), erroneously classified 7,154 as unsatisfiable, reported unknown for 3, produced an error for 562, and timed out on 84.

CVC4 timed out on 2,388 problems, but produced no errors and no unknown answers. For the problems that CVC4 timed out on, Z3-STR classified 201 as unsatisfiable, returned an error for 34 and produced solutions for the remaining 1,339, all of which were spurious. KALUZA classified 2,041 as unsatisfiable and returned an error on the rest.

These results provide strong evidence that CVC4's string solver is sound. They also provide evidence that unsat answers from Z3-STR and KALUZA for problems on which CVC4 times out cannot be trusted. They also show that CVC4's string solver answers sat more often than both Z3-STR and KALUZA, providing a correct solution in each case. Thus, it is overall the best tool for both satisfiable and unsatisfiable problems.

Moving to run time performance, a comparison with KALUZA is not very meaningful because of its high unreliability and the unverifiability of its sat answers. In principle, the same could be said of Z3-STR due to its refutation unsoundness.[14]

[14] Z3-STR could be faster and time out less often simply because it unduly prunes search space.

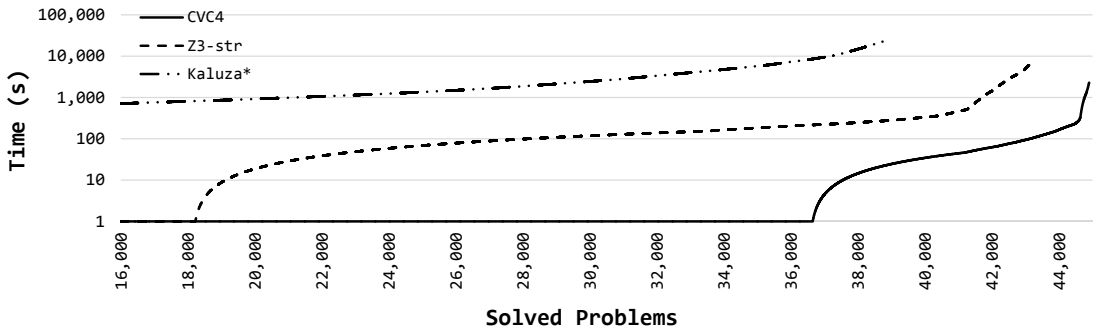

Figure 3.19. Runtime comparison of CVC4, Z3-STR and the amended KALUZA. Times are in seconds.

However, an analysis of our detailed results shows that CVC4 has nonetheless better runtime performance overall. This can be easily seen from the cactus plot in Figure 3.19, which shows for each of the three systems how many non-provably incorrect benchmarks it cumulatively solves within a certain amount of time.

3.5 Summary

In this chapter, we discussed our calculus for handling word equations with length constraints. Our novel approach solves problems from this theory directly, without reduction to other problems (e.g., automata or bit-vectors in a conventional way). We have implemented this calculus in a state-of-the-art SMT solver CVC4. By utilizing the DPLL(T) architecture, as well as other off-the-shelf background theory engines (e.g CVC4), our string solver can solve string constraints together with other theories. Furthermore, our initial results indicate that our approach outperforms the other existing string solvers in terms of correctness, precision and performance.

Besides the calculus, we provided proofs for refutation soundness and solution soundness of our calculus. Together with the *fair* strategy, we showed the solution

completeness for the theory of unbounded strings with length constraints. Moreover, if the input is in the acyclic form, we showed that our calculus is a decision procedure for this fragment.

In addition, we discussed our refinements when propagation is applied in the case of string constant splitting. Also, we provide our extension for other string manipulating functions, which are frequently used in security vulnerability detection.

CHAPTER 4

SOLVING MEMBERSHIP AND LENGTH CONSTRAINTS

In this chapter, we present algebraic techniques for solving membership and length constraints. They follow a more general approach than the one we discussed in Chapter 3 for handling membership constraints. The major difference is that we replace the procedure of unrolling the Kleene stars (which has no guarantee to be terminating when the problem is unsatisfiable) with a more sophisticated approach.

In the new approach, we break a regular expression into sub-expressions without introducing additional fresh variables. Besides expression splitting, we associate a regular expression with a set of linear arithmetic constraints that represent all possible lengths of strings in the language of the regular expression. On inputs with only concrete regular membership and length constraints, this approach completes our model search algorithm. Indeed, it is a decision procedure for this fragment.

In addition, we extend our procedure for handling symbolic membership constraints. With the extension of symbolic expressions, the termination depends on the success of solving non-linear arithmetic constraints over a particular fragment.

Organization. In Section 4.1, we present our calculus for solving concrete regular membership constraints with additional length constraints, and prove that our approach is a decision procedure for this fragment. In Section 4.2, we extend our calculus for handling symbolic regular membership constrains, and for solving negative membership constraints lazily. Finally, a summary of this chapter is given in

Section 4.3.

4.1 Calculus for T_{RL}

Before discussing our calculus, we briefly introduce the satisfiability problem for constraints containing membership constraints, which is an extension of Section 3.1. A *membership* is a predicate which takes two arguments: a string and an expression denoting a language. In this section, an expression refers to a concrete regular expression as defined in Section 2.6. In Section 4.2.1, we extend of our calculus for handling constraints over symbolic regular expressions. Throughout this chapter, we consider derivations without the rule **R-Star**.

As described in Section 2.3, a concrete regular expression contains string constants only. The *core* regular expressions are *well-sorted* terms built upon singletons (set), the *empty set* (rempty), and the *all character* set (allchars), as well as four regular expression operations: *concatenation* (rcon), *alternation* (union), *intersection* (inter), and *Kleene star* (star).

Assumption 3 In this section, we assume that the polarity of membership constraints are all positive. If a negative membership constraint appears in the set of membership constraints R in a configuration c, we always rewrite it to a positive one by some standard algorithm (e.g., [37]) during the preprocessing, i.e.,

$$\neg\mathsf{in}(s, R) \rightarrow \mathsf{in}(s, R^c), \tag{4.1}$$

where R^c is the complement of R. □

Although it is known that regular languages are *closed* under the complemen-

tation (see Proposition 9 in Chapter 2), the complexity of computing the complement of a regular expression is very high [37]. Therefore, in Section 4.2.1, we will introduce a new lazy approach to handle negative membership constraints without applying the complement operation.

Assumption 4 We assume that the component R in a configuration only contains regular expressions from the core language. □

We overload the function $\mathcal{L}(\_)$ that takes a term of Lan and returns a set of strings throughout this section.

4.1.1 Preprocessing

We define those rewriting rules in Figure 4.1, so one could apply them to remove the additional symbols. We use R^n to represent n replications (of a regular expression R) connected by concatenation [1]. The preprocessing happens at the very beginning, and only once. Also, all other regular expression rules in our calculus do not introduce any of the regular expression operations form Figure 4.1.

In the following, we describe these rules in detail. The Formula 4.2 reflects the semantics of the regular expression *option* operation: $\mathsf{in}(s, \mathsf{opt}(R))$ is true iff either the string term s is the empty string or its instance is in $\mathcal{L}(R)$.

The rule Formula 4.3 reflects the semantics of the regular expression *plus* operation: if $\mathcal{L}(R)$ does not contain the empty string, then the string cannot be the empty string; otherwise, it is equivalent to $\mathcal{L}(\mathsf{star}(R))$.

---

[1] The symbol R^n is a shorthand for $\mathsf{rcon}(\underbrace{R, R, \cdots, R}_{n \text{ copies}})$.

$$\mathsf{in}(s, \mathsf{opt}(R)) \to s \approx \epsilon \vee \mathsf{in}(s, R) \tag{4.2}$$

$$\mathsf{in}(s, \mathsf{plus}(R)) \to \mathsf{in}(s, \mathsf{star}(R)) \wedge \delta(R) \Rightarrow s \not\approx \epsilon \tag{4.3}$$

$$\mathsf{in}(s, \mathsf{range}(c_1, c_2)) \to s \approx c_1 \vee \cdots \vee s \approx c_2 \tag{4.4}$$

$$\mathsf{in}(s, \mathsf{range}(c_1, c_3)) \to \mathsf{false} \tag{4.5}$$

$$\mathsf{in}(s, \mathsf{loop}(R, n, m)) \to \mathsf{in}(s, R^n) \vee \cdots \vee \mathsf{in}(s, R^m) \tag{4.6}$$

$$\mathsf{in}(s, \mathsf{loop}(R, n)) \to \mathsf{in}(s, \mathsf{rcon}(R^n, \mathsf{star}(R))) \tag{4.7}$$

Figure 4.1. Membership term preprocessing rules, where n, m are natural numbers ($n \leq m$), c_1, c_2, c_3 are characters ($c_1 \preceq_{lex} c_2$ and $c_3 \prec_{lex} c_1$), R is a concrete regular expression, and δ is the delta function described later.

The Formula 4.4 reflects the semantics of the regular expression range operation: for any model $\mathcal{M}$, $\mathcal{M}[\![s]\!]$ must be a character between c_1 and c_2 in the lexicographic order; otherwise, Formula 4.5 will be applied.

The rule Formula 4.6 reflects the semantics of the regular expression loop operation: the string s must be in the language of R^i ($n \leq i \leq m$). Similarly, Formula 4.7 indicates that the string must be in the language of R^i ($n \leq i$).

4.1.2 Normalization

Similarly to Figure 3.1, we introduce a set of regular expression term rewriting rules in Figure 4.2.

Assumption 5 We also assume that every term in the initial configuration is reduced with respect to the rewrite rules in both Figure 3.1 and Figure 4.2. □

The rewriting with respect to the rewriting rules in both Figure 3.1 and Figure 4.2 can be shown to be terminating and confluent modulo the axioms of arithmetic by measuring the decrease of total symbols in a regular expression after each rewriting. Since the left hand side is equivalent to the right hand side of each rule, it is easy to show the correctness of these rules.

Note that an arbitrary *non-flattened* [2] regular expression can be converted to a *flattened* regular expression by the rules in Figure 4.2. For simplicity, when a concatenation (similarly an alternation, or an intersection) contains only one atomic regular expression, we remove the top-level symbol, i.e., $\mathsf{rcon}(R) \to R$, $\mathsf{union}(R) \to R$ and $\mathsf{inter}(R) \to R$, where R is an atomic regular expression.

Besides the term-level rewrite rules (in Figure 4.2) for regular expressions, we have another set of rules (in Figure 4.3) which rewrite membership constraints. We can consider them as *literal-level* rewrite rules. Note that in this section, all regular expressions contain no variable.

Lemma 24 Rules in Figure 4.3 preserve the equivalence in the theory.

Proof: $\mathsf{in}(s, \mathsf{allchars})$ is equivalent to $\mathsf{len}(s) \approx 1$ because $\mathsf{allchars}$ represents all strings of length 1. $\mathsf{in}(s, \mathsf{star}(\mathsf{allchars}))$ is equivalent to true because $\mathsf{star}(\mathsf{allchars})$ represents all strings. $\mathsf{in}(s, \mathsf{set}(t))$ is equivalent to $s \approx t$ by the definition. ■

Assumption 6 We assume that the procedure applies all these rewriting rules from both Figure 4.2 and Figure 4.3 (to all membership constraints) to completion before

[2] A non-flattened regular expression is the one with nested concatenations (or alternations or intersections) as defined in Chapter 2.

$$\mathsf{rcon}(\boldsymbol{r_1}, \mathsf{rcon}(\boldsymbol{r_2}), \boldsymbol{r_3}) \rightarrow \mathsf{rcon}(\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}) \tag{4.8}$$

$$\mathsf{rcon}(\boldsymbol{r_1}, \mathsf{set}(\epsilon), \boldsymbol{r_3}) \rightarrow \mathsf{rcon}(\boldsymbol{r_1}, \boldsymbol{r_3}) \tag{4.9}$$

$$\mathsf{rcon}(\boldsymbol{r_1}, \mathsf{rempty}, \boldsymbol{r_3}) \rightarrow \mathsf{rempty} \tag{4.10}$$

$$\mathsf{rcon}(\boldsymbol{r_1}, \mathsf{set}(s_1), \mathsf{set}(s_2), \boldsymbol{r_3}) \rightarrow \mathsf{rcon}(\boldsymbol{r_1}, \mathsf{set}(\mathsf{con}(s_1, s_2)), \boldsymbol{r_3}) \tag{4.11}$$

$$\mathsf{rcon}(r) \rightarrow r \tag{4.12}$$

$$\mathsf{rcon}() \rightarrow \mathsf{rempty} \tag{4.13}$$

$$\mathsf{union}(\boldsymbol{r_1}, \mathsf{union}(\boldsymbol{r_2}), \boldsymbol{r_3}) \rightarrow \mathsf{union}(\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}) \tag{4.14}$$

$$\mathsf{union}(\boldsymbol{r_1}, \mathsf{rempty}, \boldsymbol{r_3}) \rightarrow \mathsf{union}(\boldsymbol{r_1}, \boldsymbol{r_3}) \tag{4.15}$$

$$\mathsf{inter}(\boldsymbol{r_1}, \mathsf{inter}(\boldsymbol{r_2}), \boldsymbol{r_3}) \rightarrow \mathsf{inter}(\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}) \tag{4.16}$$

$$\mathsf{inter}(\boldsymbol{r_1}, \mathsf{rempty}, \boldsymbol{r_3}) \rightarrow \mathsf{rempty} \tag{4.17}$$

$$\mathsf{star}(\mathsf{star}(r)) \rightarrow \mathsf{star}(r) \tag{4.18}$$

$$\mathsf{star}(\mathsf{rempty}) \rightarrow \mathsf{set}(\epsilon) \tag{4.19}$$

Figure 4.2. Regular expression rewriting rules, where s_1, s_2 are terms of String, r is a term of Lan, and $\boldsymbol{r_1}$, $\boldsymbol{r_2}$, $\boldsymbol{r_3}$ are vectors of Lan terms.

applying any other reduction rule.

The word equations generated by $\mathsf{in}(s, \mathsf{set}(t)) \rightarrow s \approx t$ belong to *One-Sided* equality. By Corollary 3, our derivation is guaranteed to terminate.

4.1.3 Auxiliary Functions

The rules in our calculus rely on some condition that some expressions are computed by a set of auxiliary functions. We will introduce these functions first.

In addition, we prove the termination and correctness of each function.

$$\mathsf{in}(s, \mathsf{allchars}) \to \mathsf{len}(s) \approx 1 \tag{4.20}$$

$$\mathsf{in}(s, \mathsf{star}(\mathsf{allchars})) \to \mathsf{true} \tag{4.21}$$

$$\mathsf{in}(s, \mathsf{rempty}) \to \mathsf{false} \tag{4.22}$$

$$\mathsf{in}(s, \mathsf{set}(t)) \to s \approx t \tag{4.23}$$

$$\mathsf{in}(l, R) \to \zeta(l, R) \tag{4.24}$$

Figure 4.3. Membership term rewriting rules, where s, t are string terms and l is a string constant.

Concrete Delta Function

Similar to [21, 68, 24], we define our reduction function δ as follows: it takes a term of type Lan R, and returns either true or false, where $\mathcal{L}(R) \in \mathbb{L}_{\mathbb{C}}$. The function δ returns true iff the language represented by the regular expression contains the empty string. This function is defined inductively in Figure 4.4.

Lemma 25 For any concrete term R of sort Lan, the function of computing $\delta(R)$, with respect to the rules in Figure 4.4, terminates.

Proof: The termination of the δ function computation for an arbitrary regular expression follows measuring the fact that the total number of symbols in this regular expression decreases. ■

Lemma 26 For any concrete term R of sort Lan, $\delta(R) = \mathsf{true}$ iff $\epsilon \in \mathcal{L}(R)$.

Proof: By Lemma 25, the function terminates on any concrete term. Now, we argue the correctness by cases.

$$\delta(\mathsf{allchars}) = \mathsf{false} \tag{4.25}$$

$$\delta(\mathsf{rempty}) = \mathsf{false} \tag{4.26}$$

$$\delta(\mathsf{set}(\epsilon)) = \mathsf{true} \tag{4.27}$$

$$\delta(\mathsf{set}(l)) = \mathsf{false} \text{ if } l \not\approx \epsilon \tag{4.28}$$

$$\delta(\mathsf{rcon}(R_1, \cdots, R_n)) \text{ is } \mathsf{true} \text{ iff } \forall i \in [1 \ldots n].\delta(R_i) = \mathsf{true} \tag{4.29}$$

$$\delta(\mathsf{union}(R_1, \cdots, R_n)) \text{ is } \mathsf{true} \text{ iff } \exists i \in [1 \ldots n].\delta(R_i) = \mathsf{true} \tag{4.30}$$

$$\delta(\mathsf{inter}(R_1, \cdots, R_n)) \text{ is } \mathsf{true} \text{ iff } \forall i \in [1 \ldots n].\delta(R_i) = \mathsf{true} \tag{4.31}$$

$$\delta(\mathsf{star}(R)) = \mathsf{true} \tag{4.32}$$

Figure 4.4. Delta function for concrete regular expressions, where l is a string literal, and R_i's are concrete regular expressions.

For Formulas 4.25— 4.28, the result is immediate by the definition of regular expressions. For Formula 4.29, if $\mathsf{rcon}(R_1, \cdots, R_n)$ contains the empty string, then each must contain the empty string. For Formula 4.30, if $\mathsf{union}(R_1, \cdots, R_n)$ contains the empty string, then one of them must contain the empty string. For Formula 4.31, if $\mathsf{inter}(R_1, \cdots, R_n)$ contains the empty string, then each must contain the empty string. For Formula 4.32, by definition, $\mathsf{star}(R)$ contains the empty string. ■

Concrete Testing Function

We define a function ζ that takes a string literal l and a term of Lan R, and returns either true or false, where $\mathcal{L}(R) \in \mathbb{L}_{\mathbb{C}}$. This function is used to check whether a string constant l is in the language generated by the concrete regular expressions R. The ζ function returns true if the string constant is in the regular language; otherwise,

$$\zeta(l, \mathsf{rempty}) = \mathsf{false} \tag{4.33}$$

$$\zeta(l, \mathsf{set}(l)) = \mathsf{true} \tag{4.34}$$

$$\zeta(l_1, \mathsf{set}(l_2)) = \mathsf{false} \text{ if } l_1 \not\approx l_2 \tag{4.35}$$

$$\begin{aligned}\zeta(\mathsf{con}(c_1, \boldsymbol{c}_2), \mathsf{rcon}(R_1, R_2)) = \mathsf{true} \text{ iff} \\ \delta(R_1) = \mathsf{true} \text{ and } \zeta(\mathsf{con}(c_1, \boldsymbol{c}_2), R_2) = \mathsf{true}, \text{ or} \\ \zeta(c_1, R_1) = \mathsf{true} \text{ and } \zeta(\mathsf{con}(\boldsymbol{c}_2), R_2) = \mathsf{true}, \text{ or} \\ \zeta(\mathsf{con}(c_1, \boldsymbol{c}_2), R_1) = \mathsf{true} \text{ and } \delta(R_2)) = \mathsf{true}\end{aligned} \tag{4.36}$$

$$\zeta(l, \mathsf{union}(R_1, R_2)) = \mathsf{true} \text{ iff either } \zeta(l, R_1) = \mathsf{true} \text{ or } \zeta(l, R_2) = \mathsf{true} \tag{4.37}$$

$$\zeta(l, \mathsf{inter}(R_1, R_2)) = \mathsf{true} \text{ iff both } \zeta(l, R_1) = \mathsf{true} \text{ and } \zeta(l, R_2) = \mathsf{true} \tag{4.38}$$

$$\begin{aligned}\zeta(\mathsf{con}(c_1, \boldsymbol{c}_2), \mathsf{star}(R)) = \mathsf{true} \text{ iff} \\ \zeta(c_1, R) = \mathsf{true} \text{ and } \zeta(\mathsf{con}(\boldsymbol{c}_2), \mathsf{star}(R)) = \mathsf{true}, \text{ or} \\ \zeta(\mathsf{con}(c_1, \boldsymbol{c}_2), R) = \mathsf{true}\end{aligned} \tag{4.39}$$

$$\zeta(\epsilon, \mathsf{star}(R)) = \mathsf{true} \tag{4.40}$$

Figure 4.5. Testing function for concrete regular expressions, where l, l_1, l_2 are string literals, R, R_1, R_2 are regular expressions, c_1 is a character, and $\boldsymbol{c}_2$ is a vector of characters.

it returns false. The ζ function is shown in Figure 4.5.

The ζ function applies recursively and at each step it consumes one or more symbols.

Lemma 27 For any concrete term R of sort Lan, the function of computing $\zeta(R)$, with respect to the rules in Figure 4.5, terminates.

Proof: The termination of the function ζ always follows the fact that the total number of symbols of the arguments in both the string and the regular expression

decreases after each step. ■

Lemma 28 For any concrete term R of sort Lan, and for any string literal l, s.t. $\zeta(l, R) = \mathsf{true}$ iff $l \in \mathcal{L}(R)$.

Proof: By Lemma 27, the function terminates on any concrete term. Now, we show the correctness of this function.

This function maintains the semantics of regular expression. For Formulas 4.33— 4.35, the proof is immediate by the definition. For Formula 4.36, if a string constant is in the language of a regular expression with top symbol rcon, then this string can be divided into two substrings and each of them is in the language of a partitioned expression. Note that a regular expression R is partitioned into two regular expression R_1 and R_2 with respect to concatenation if $\mathcal{L}(R) = \{s \cdot t \mid s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\}$. For Formulas 4.37— 4.38, the proof follows the definition. For Formula 4.39, if a string constant is in the language of a regular expression with the top symbol star (e.g., $\mathsf{star}(R)$), then this string can be divided into two substrings so that the first substring is in the language of R and the second one is in the language of $\mathsf{star}(R)$. The empty string is always in the language of $\mathsf{star}(R)$, which is reflected in Formula 4.40. ■

Concrete Derivative Function

We use $\{\partial_c\}_{c \in \Sigma}$ to denote the family of the concrete *derivative* functions. The definition of derivatives is based on Brzozowski's work [21, 68, 24]. Further extension to symbolic expressions will be presented in Section 4.2.1.

$$\partial_c(\mathsf{rempty}) = \mathsf{rempty} \tag{4.41}$$

$$\partial_c(\mathsf{set}(\epsilon)) = \mathsf{rempty} \tag{4.42}$$

$$\partial_c(\mathsf{set}(\mathsf{con}(c, s))) = \mathsf{set}(s) \tag{4.43}$$

$$\partial_c(\mathsf{set}(\mathsf{con}(c', s))) = \mathsf{rempty}, \text{where } c \neq c' \tag{4.44}$$

$$\partial_c(\mathsf{rcon}(R_1, R_2)) = \mathsf{rcon}(\partial_c(R_1), R_2) \text{ if } \delta(R_1) = \mathsf{false} \tag{4.45}$$

$$\partial_c(\mathsf{rcon}(R_1, R_2)) = \mathsf{union}(\mathsf{rcon}(\partial_c(R_1), R_2), \partial_c(R_2)) \text{ if } \delta(R_1) = \mathsf{true} \tag{4.46}$$

$$\partial_c(\mathsf{union}(R_1, R_2)) = \mathsf{union}(\partial_c(R_1), \partial_c(R_2)) \tag{4.47}$$

$$\partial_c(\mathsf{inter}(R_1, R_2)) = \mathsf{inter}(\partial_c(R_1), \partial_c(R_2)) \tag{4.48}$$

$$\partial_c(\mathsf{star}(R)) = \mathsf{rcon}(\partial_c(R), \mathsf{star}(R)) \tag{4.49}$$

Figure 4.6. Derivative function for concrete regular expression, where c, c' are characters, s is a string term, and R, R_1, R_2 are regular expressions.

Let c be a character in Σ. We define the function $\partial_c : \mathsf{Lan} \rightarrow \mathsf{Lan}$ as the rules in Figure 4.6. The function ∂_c takes a concrete regular expression R_1, and returns a new regular expression R_2 where for every string in the language of R_1 beginning with the character c, the first character is consumed. In other words, the regular expression R_2 represents the language $\{u \mid c \cdot u \in \mathcal{L}(R_1)\}$.

The derivative function gives us a way to handle regular expression operations without reduction to automata, as the conventional method of processing regular expressions suggests. In other words, the function ∂ gives us a basis to build algebra-based rules. The derivative function is shown in Figure 4.6.

Lemma 29 For any concrete term R of Lan, and for any character c in Σ, s.t. the

function of computing $\mathcal{L}(\partial_c R)$, with respect to the rules in Figure 4.6, terminates.

Proof: Since for each step the algorithm consumes some symbols in the regular expression, and there are finitely many symbols in a regular expression, the algorithm will terminate on any regular expression. ■

Theorem 7 *For any concrete regular term R of* Lan*, and for any character c in Σ, s.t. $\mathcal{L}(\partial_c R) = \{s \mid c \cdot s \in \mathcal{L}(R)\}$.*

Proof: By Lemma 29, the function $\mathcal{L}(\partial_c R)$ on any regular expression, for any character. Now, we show that $\mathcal{L}(\partial_c R) = \{s \mid c \cdot s \in \mathcal{L}(R)\}$.

Let R be an arbitrary concrete regular expression, and c be an arbitrary character in Σ. We prove by induction, following Formulas 4.41— 4.49 for finding a regular expression with respect to a single character.

Prove by induction on the structure of regular expressions. The base cases, Formulas 4.41— 4.44 hold trivially by the definition of a regular expression. Now, we show it holds for step cases.

For Formula 4.45, since $\delta(R_1) = \mathsf{false}$, we have :

$$\begin{aligned}
\mathcal{L}(\mathsf{rcon}(\partial_c(R_1), R_2)) &= \{s \mid c \cdot s \in \mathcal{L}(R_1)\} \cdot \mathcal{L}(R_2) \\
&= \{s \cdot t \mid c \cdot s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\} \\
&= \{s \mid c \cdot s \in \mathcal{L}(\mathsf{rcon}(R_1, R_2))\} \\
&= \mathcal{L}(\partial_c(\mathsf{rcon}(R_1, R_2))).
\end{aligned}$$

For Formula 4.46, since $\delta(R_1) = \mathsf{true}$, we have :

$$\begin{aligned}
\mathcal{L}(\mathsf{union}(\mathsf{rcon}(\partial_c(R_1), R_2), \partial_c(R_2))) &= \mathcal{L}(\mathsf{rcon}(\partial_c(R_1), R_2)) \cup \mathcal{L}(\partial_c(R_2)) \\
&= \{s \mid c \cdot s \in \mathcal{L}(R_1)\} \cdot \mathcal{L}(R_2) \cup \mathcal{L}(\partial_c(R_2)) \\
&= \{s \cdot t \mid c \cdot s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\} \cup \mathcal{L}(\partial_c(R_2)) \\
&= \{s \mid c \cdot s \in \mathcal{L}(\mathsf{rcon}(R_1, R_2))\} \\
&= \mathcal{L}(\partial_c(\mathsf{rcon}(R_1, R_2))).
\end{aligned}$$

For Formula 4.47, we have :

$$\begin{aligned}
\mathcal{L}(\mathsf{union}(\partial_c(R_1), \partial_c(R_2))) &= \{s \mid c \cdot s \in \mathcal{L}(R_1)\} \cup \{s \mid c \cdot s \in \mathcal{L}(R_2)\} \\
&= \{s \mid c \cdot s \in \mathcal{L}(R_1) \cup \mathcal{L}(R_2)\} \\
&= \{s \mid c \cdot s \in \mathcal{L}(\mathsf{union}(R_1, R_2))\} \\
&= \mathcal{L}(\partial_c(\mathsf{union}(R_1, R_2))).
\end{aligned}$$

Similar to the proof of Formula 4.48.

For Formula 4.49, we have :

$$\begin{aligned}
\mathcal{L}(\mathsf{rcon}(\partial_c(R), \mathsf{star}(R))) &= \emptyset \ \cup \ \{s \mid c \cdot s \in \mathcal{L}(\mathsf{rcon}(R, \mathsf{star}(R)))\} \\
&= \{s \mid c \cdot s \approx \epsilon\} \ \cup \ \{s \mid c \cdot s \in \mathcal{L}(\mathsf{rcon}(R, \mathsf{star}(R)))\} \\
&= \{s \mid c \cdot s \in \mathcal{L}(\mathsf{union}(\mathsf{set}(\epsilon), R, R^2, \cdots))\} \\
&= \mathcal{L}(\partial_c(\mathsf{star}(R))).
\end{aligned}$$

Since all symbols belong to the theory of regular expressions, the returned expression is also a regular expression. ■

The derivative function can be extended to accept a sequence of characters. Given a string constant $l = c_0 \cdots c_{n-1}$ and a regular expression R, the string derivative function is defined as follows:

$$\partial_l R = \partial_{c_{n-1}} \cdots \partial_{c_0} R. \tag{4.50}$$

Corollary 5 For any concrete term R of Lan, and for any non-empty string literal l, s.t. $l \in \mathcal{L}(R)$ iff $\epsilon \in \mathcal{L}(\partial_l R)$.

Proof: We prove by induction on the structure of a regular expression. The base case can be proved by Theorem 7, that is, if a character c is in a regular expression R, then $\mathcal{L}(\partial_c R)$ must contains the empty string. We assume that Formula 4.50 holds for smaller regular expressions. We now show that when $l = c_0 \cdots c_n$, the formula still holds. $c_0 \cdots c_n \in \mathcal{L}(R)$ iff $c_1 \cdots c_n \in \mathcal{L}(\partial_{c_0} R)$. By hypothesis, we have $c_1 \cdots c_n \in \mathcal{L}(\partial_{c_0} R)$ iff $\epsilon \in \mathcal{L}(\partial_{c_1 \cdots c_n} R)$. Therefore, we can conclude that $l \in \mathcal{L}(R)$ iff the empty string is in $\mathcal{L}(\partial_l R)$. ■

First Characters Function

Now we define the first characters function α. It takes a term of Lan R and returns a set of characters where any string in the language of R must begin with one of the characters in the set, i.e., $\alpha(R) = \{c \mid c \cdot l \in \mathcal{L}(R), \text{ for some } l\}$. The first characters function is shown in Figure 4.7.

This function is used for the intersection algorithm. It is an optional function, but it helps reducing unnecessary computation.

$$\alpha(\mathsf{rempty}) = \emptyset \tag{4.51}$$

$$\alpha(\mathsf{set}(\epsilon)) = \emptyset \tag{4.52}$$

$$\alpha(\mathsf{set}(\mathsf{con}(c, s))) = \{c\} \tag{4.53}$$

$$\alpha(\mathsf{allchars}) = \Sigma \tag{4.54}$$

$$\alpha(\mathsf{rcon}(R_1, \cdots, R_n)) = \alpha(R_1) \cup \cdots \cup \alpha(R_n) \tag{4.55}$$

$$\alpha(\mathsf{union}(R_1, \cdots, R_n)) = \alpha(R_1) \cup \cdots \cup \alpha(R_n) \tag{4.56}$$

$$\alpha(\mathsf{inter}(R_1, \cdots, R_n)) = \alpha(R_1) \cap \cdots \cap \alpha(R_n) \tag{4.57}$$

$$\alpha(\mathsf{star}(R)) = \alpha(R) \tag{4.58}$$

Figure 4.7. First characters function for concrete regular expressions, where c is a character, and R, R_i's are regular expressions.

Lemma 30 The function α, with respect to the rules in Figure 4.7, terminates on any regular expression R. We have $\alpha(R) = \{c \mid c \cdot l \in \mathcal{L}(R), \text{ for some } l\}$.

Proof: The correctness proof of this function follows the definition of regular expressions. ■

Intersection of Concrete Regular Expressions

In our calculus, we use a derivative-based algorithm to compute the intersection of two concrete regular expressions. To start with the intersection algorithm used in our calculus, we first introduce several lemmas which follow the step of Theorem 7.

Lemma 31 Let R be an arbitrary regular expression. R can be normalized to the

form:

$$\mathsf{union}(\mathsf{ite}(\delta(R), \mathsf{set}(\epsilon), \mathsf{rempty}),$$

$$\bigcup_{c \in \Sigma} (\mathsf{rcon}(\mathsf{set}(c), \partial_c R))).$$

Proof: Any regular expression may or may not contain the empty string, and the first part $\mathsf{ite}(\delta(R), \mathsf{set}(\epsilon), \mathsf{rempty})$ takes care of it.

For any non-empty strings in the language of R, it must begin with a character (from Σ), say c. By the definition of the derivative function (in Theorem 7) $\mathcal{L}(\partial_c R) = \{s \mid c \cdot s \in \mathcal{L}(R)\}$, we know that all suffixes of the strings (in the language of R) beginning with c, can be represented as $\mathsf{rcon}(\mathsf{set}(c), \partial_c R)$. By Theorem 7, if a regular language does not contain a string beginning with c, $\partial_c R$ returns rempty. Thus, the non-empty strings in $\mathcal{L}(R)$ can be taken care of by $\bigcup_{c \in \Sigma}(\mathsf{rcon}(\mathsf{set}(c), \partial_c R))$.

Therefore, any regular expression can be normalized to the above form. ■

Note that there is a unique normal form for every regular expression obtained by applying the derivative algorithm. Therefore, Lemma 31 have our naïve version of the intersection algorithm.

Lemma 32 (Naïve Intersection) Let R_1 and R_2 be two arbitrary regular expressions. The intersection of R_1 and R_2, $\mathsf{inter}(R_1, R_2)$, can be denoted as

$$\mathsf{union}(\ \mathsf{ite}(\delta(R_1) \wedge \delta(R_2),\ \mathsf{set}(\epsilon),\ \mathsf{rempty}),$$

$$\bigcup_{c \in \Sigma} (\ \mathsf{rcon}(\mathsf{set}(c), \mathsf{inter}(\partial_c R_1, \partial_c R_2))\ )\ ).$$

Proof: The proof is immediate by converting these two regular expressions R_1 and

R_2 into their normal forms (shown in Lemma 31), and then converting the intersection operations into conjunctions. ■

The formula in Lemma 32 can be used as an algorithm for computing the intersection of two regular expressions. However, this naïve version of intersection algorithm is not terminating in general. For instance, $\partial_a(\mathsf{inter}(\mathsf{star}(\mathsf{set}(a)), \mathsf{star}(\mathsf{set}(a))))$ is equal to $\mathsf{inter}(\mathsf{star}(\mathsf{set}(a)), \mathsf{star}(\mathsf{set}(a)))$, which leads to an unproductive computation. To counter this, we introduce more properties.

Note that if the languages of two regular expressions are equal, we say these two regular expressions recognize the same language. Two regular expressions recognizing the same language, may not have the same syntactic form.

Example 10 The regular expression $\mathsf{rcon}(\mathsf{set}(aa), \mathsf{star}(\mathsf{set}(aa)))$ and the regular expression $\mathsf{rcon}(\mathsf{star}(\mathsf{set}(aa)), \mathsf{set}(aa))$ denote the same language. □

Let R be a regular expression. By Theorem 7, we have $\partial_s R$ is a regular expression, for any arbitrary string s. We use ∂R to denote a set of languages that $\partial_s R$ can generate (for all string s), i.e., $\mathcal{L}(\partial R) = \{\mathcal{L}(\partial_s R) \mid s \in \mathcal{A}^*\}$. We call this set the derivative set of R. We now show the cardinality of this set (denoted as $|\partial R|$) is finite.

Lemma 33 Let R be an arbitrary regular expression, the cardinality of the derivative set of R is finite.

Proof: We show by induction on the number of regular expression operators in the regular expression R.

Base Case: If there is no operator in R, we have three cases.

- If $R = \mathsf{rempty}$, we have $\partial_c \mathsf{rempty} = \mathsf{rempty}, \forall c \in \Sigma$. We say $|\partial \mathsf{rempty}| = 1$.
- If $R = \mathsf{set}(\epsilon)$, we have $\partial_c \mathsf{set}(\epsilon) = \mathsf{rempty}, \forall c \in \Sigma$. We say $|\partial \mathsf{set}(\epsilon)| = 1$.
- If $R = \mathsf{set}(s), s \neq \epsilon$, we have $\partial_a \mathsf{set}(\mathsf{set}(\mathsf{con}(a, t))) = \mathsf{set}(t)$ and $\partial_c \mathsf{set}(\mathsf{set}(s)) = \mathsf{rempty}, \forall c \in \Sigma / \{a\}$. We say $|\partial \mathsf{set}(s)| = 2$.

Induction Hypothesis: Given a regular expression R and there are N operators in R, then $|\partial R|$ is finite.

Step Case: Assume a regular expression R contains $N+1$ operators. We show the property hold by proving the following four cases. Note that we only need to show that only finitely many languages are generated by applying derivative function to one character. The induction hypothesis will take place after that.

- Assume $R = \mathsf{rcon}(R_1, R_2)$. If $\delta(R_1) = \mathsf{false}$, by definition, we have $\partial_c \mathsf{rcon}(R_1, R_2)$ is equal to $\mathsf{rcon}(\partial_c R_1, R_2)$. Thus, $|\partial \mathsf{rcon}(R_1, R_2)| = |\partial R_1|$. By Induction Hypothesis, this number is finite.

  If $\delta(R_1) = \mathsf{true}$, by definition, we have $\partial_c \mathsf{rcon}(R_1, R_2)$ is equal to the union of $\mathsf{rcon}(\partial_c R_1, R_2)$ and $\partial_c R_2$. Thus, $|\partial \mathsf{rcon}(R_1, R_2)| \leq |\partial R_1| \times |\partial R_2|$. By Induction Hypothesis, this number is finite.

- Assume $R = \mathsf{union}(R_1, R_2)$. By definition, we have $\partial_c \mathsf{union}(R_1, R_2)$ is equal to $\mathsf{union}(\partial_c R_1, \partial_c R_2)$. Thus, $|\partial \mathsf{union}(R_1, R_2)| \leq |\partial R_1| \times |\partial R_2|$. By Induction Hypothesis, this number is finite.

- Assume $R = \mathsf{inter}(R_1, R_2)$. By definition, we have $\partial_c \mathsf{inter}(R_1, R_2)$ is equal to $\mathsf{inter}(\partial_c R_1, \partial_c R_2)$. Thus, $|\partial \mathsf{inter}(R_1, R_2)| \leq |\partial R_1| \times |\partial R_2|$. By Induction Hypothesis, this number is finite.

- Assume $R = \mathsf{star}(R_1)$. By definition, we have $\partial_c \mathsf{star}(R_1)$ is equal to the concatenation of $\partial_c R_1$ and $\mathsf{star}(R_1)$. Thus, $|\partial \mathsf{star}(R_1)| = |\partial R_1|$. By Induction Hypothesis, this number is finite.

It concludes that $|\partial R|$ is finite, for any R. ■

Based on Lemmas 32 and 33, we provide another intersection algorithm. This algorithm was originally described in [84, 85] and we made some modifications according to our needs. Notice that deciding whether a language is equal to the intersection of two other languages is of *NON-ELEMENTARY* complexity [4]. Moreover, scanning over all characters over larger alphabet is impractical (e.g., UTF-8 contains over one million characters). Because of the expensive complexity in computing an intersection, most algorithms use an approximation for such computation (e.g., [15, 78]). Although our algorithm does not use approximation, due to the theoretical complexity, we cannot avoid exponential growth in general.

Any regular expression R can be expressed by a pair $\langle R_1, R_2 \rangle$, namely *co-RE*, where the first regular expression is for the repeating part and the second one is for the residue, i.e., $R = \mathsf{rcon}(\mathsf{star}(R_1), R_2)$. The function $rcov : \langle \mathsf{Lan}, \mathsf{Lan} \rangle \rightarrow \mathsf{Lan}$ converts a co-RE (a pair of regular expressions) to its corresponding regular expression.

A regular expression is called *elementary* if it is either rempty or $\mathsf{set}(\epsilon)$. When computing an intersection of two non-elementary regular expressions (e.g., R_1 and

R_2 where neither is elementary), a temporary placeholder V_{R_1,R_2} may be used to represent the actual value of the intersection.

The function $cov_p : \mathsf{Lan} \to \langle \mathsf{Lan}, \mathsf{Lan} \rangle$ converts a regular expression to a co-RE, based on the placeholder p. This function is defined as follows:

- $cov_p(\mathsf{rempty}) = \langle \mathsf{rempty}, \mathsf{rempty} \rangle$
- $cov_p(p) = \langle \mathsf{set}(\epsilon), \mathsf{rempty} \rangle$
- $cov_p(R) = \langle \mathsf{set}(\epsilon), R \rangle$ if p does not occur in R
- $cov_p(\mathsf{rcon}(R_1, R_2)) = \langle \mathsf{rcon}(R_1, R_3), R_4 \rangle$ if p occurs in R_2 but not in R_1, where $cov_p(R_2) = \langle R_3, R_4 \rangle$
- $cov_p(\mathsf{union}(R_1, R_2)) = \langle \mathsf{union}(R_3, R_5), \mathsf{union}(R_4, R_6) \rangle$, where $cov_p(R_1) = \langle R_3, R_4 \rangle$ and $cov_p(R_2) = \langle R_5, R_6 \rangle$

The function $\pi'_C : \langle \mathsf{Lan}, \mathsf{Lan} \rangle \to \mathsf{Lan}$ computes the intersection with the cache C, where C is a set of placeholders.

- $\pi'_C(R_1, R_2) = \mathsf{rempty}$ if either R_1 or R_2 is rempty
- $\pi'_C(R, R) = R$
- $\pi'_C(R_1, \mathsf{set}(\epsilon)) = \mathsf{set}(\epsilon)$, where $\delta(R_1) = \mathsf{true}$
- $\pi'_C(R_1, \mathsf{set}(\epsilon)) = \mathsf{rempty}$, where $\delta(R_1) = \mathsf{false}$
- $\pi'_C(\mathsf{set}(\epsilon), R_2) = \mathsf{set}(\epsilon)$, where $\delta(R_2) = \mathsf{true}$

- $\pi'_C(\mathsf{set}(\epsilon), R_2) = \mathsf{rempty}$, where $\delta(R_2) = \mathsf{false}$

- $\pi'_C(R_1, R_2) = V_{R_1,R_2}$ if $V_{R_1,R_2} \in C$

- $\pi'_C(R_1, R_2) = rcov(cov_{V_{R_1,R_2}}(\mathsf{union}(\mathsf{set}(\epsilon),$
  $\bigcup_{c\in\alpha(R_1)\cap\alpha(R_2)} \mathsf{rcon}(\mathsf{set}(c), \pi'_{C\cup\{V_{R_1,R_2}\}}(\partial_c(R_1), \partial_c(R_2)))))$
  if $V_{R_1,R_2} \notin C$ and $\delta(R_1) = \mathsf{true}$ and $\delta(R_2) = \mathsf{true}$

- $\pi'_C(R_1, R_2) = rcov(cov_{V_{R_1,R_2}}($
  $\bigcup_{c\in\alpha(R_1)\cap\alpha(R_2)} \mathsf{rcon}(\mathsf{set}(c), \pi'_{C\cup\{V_{R_1,R_2}\}}(\partial_c(R_1), \partial_c(R_2))))$
  if $V_{R_1,R_2} \notin C$ and either $\delta(R_1) = \mathsf{false}$ or $\delta(R_2) = \mathsf{false}$

Therefore, the intersection $\pi(R_1, R_2) = \pi'_\emptyset(R_1, R_2)$.

Theorem 8 *Given two arbitrary regular expressions, the intersection function terminates and correctly returns a regular expression whose language is the intersection of the two.*

Proof: The correctness is given by Lemma 32. The termination proof is a consequence of Lemma 33. More detailed proof is provided in [59, 6]. ■

Splitting Function

Given a membership constraint $\mathsf{in}(t, R)$, where t is a concatenation of two string terms $t \approx \mathsf{con}(s_1, s_2)$, one intuitive way for finding a model for the concatenated strings is to break the regular expression into two regular expressions, so that the first string is in the language of the first sub-expression and the second string is in the language of the second sub-expression.

This idea has some similarities with the normalization idea in Chapter 3. Solving a membership constraint using this approach requires consideration of all possible splits. In other words, if a membership constraint $\mathsf{in}(\mathsf{con}(s_1, s_2), R)$ is unsatisfiable, it is unsatisfiable for all splits. If it is satisfiable, then it is satisfiable for at least one split. Formally, we have the following equation:

$$\mathsf{in}(\mathsf{con}(s_1, s_2), R) \Longleftrightarrow$$
$$\exists r_1, r_2 : \mathsf{Lan}.\ R \approx \mathsf{rcon}(r_1, r_2) \wedge \mathsf{in}(s_1, r_1) \wedge \mathsf{in}(s_2, r_2). \tag{4.59}$$

One approach for utilizing this idea is to represent R as an automaton and split the automaton [1]. Obviously, every regular expression can be associated with an automaton by Kleene's Theorem in Section 2.3. Also, we can always break a Finite-state Automaton (FA) into two pieces. The general algorithm for splitting an FA can be described as follows:

Let l_s, l_t be two string literals, R be a concrete regular expression, $A = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ be a DFA that recognizes $\mathcal{L}(R)$, then we have :

$$l_s \cdot l_t \in \mathcal{L}(R) \text{ iff } \bigvee_{q \in Q} l_s \in \mathcal{L}(R^l_q) \wedge l_t \in \mathcal{L}(R^r_q), \tag{4.60}$$

where R^l_q is a regular expression whose language is recognized by $\langle Q, \mathcal{A}, \delta, q_0, q\rangle$ and R^r_q is a regular expression whose language is recognized by $\langle Q, \mathcal{A}, \delta, q, F\rangle$.

Because the number of states in a DFA is finite, the number of literals in the disjunction is finite. Formula 4.60 provides a theoretical basis for regular expression splitting. However, the computation is very expensive because :

- conversion from a regular expression to a DFA is expensive, and

- conversion from a DFA to a regular expression is also expensive.

Therefore, a novel idea for splitting any regular expression without consulting to a DFA is of a high interest.

We define our splitting function β that maps a regular expression to a set of regular expression pairs in Figure 4.8.

$$\beta(\mathsf{rempty}) = \emptyset \tag{4.61}$$

$$\begin{aligned}\beta(\mathsf{set}(c_0 \cdots c_{n-1})) = \{\ & \langle \mathsf{set}(\epsilon), \mathsf{set}(c_0 \cdots c_{n-1}) \rangle, \\ & \langle \mathsf{set}(c_0), \mathsf{set}(c_1 \cdots c_{n-1}) \rangle, \\ & \vdots \\ & \langle \mathsf{set}(c_0 \cdots c_{n-1}), \mathsf{set}(\epsilon) \rangle\ \}\end{aligned} \tag{4.62}$$

$$\begin{aligned}\beta(\mathsf{rcon}(R_1, \cdots, R_n)) = \{\ & \langle \mathsf{rcon}(R_1, \cdots, R_{i-1}, r_1), \mathsf{rcon}(r_2, R_{i+1}, \cdots, R_n) \rangle \\ & \mid \langle r_1, r_2 \rangle \in \beta(R_i), \forall i \in [1, n]\ \}\end{aligned} \tag{4.63}$$

$$\beta(\mathsf{union}(R_1, \cdots, R_n)) = \beta(R_1) \cup \cdots \cup \beta(R_n) \tag{4.64}$$

$$\beta(\mathsf{inter}(R_1, \cdots, R_n)) = \beta(\pi(R_1, \cdots, R_n)) \tag{4.65}$$

$$\begin{aligned}\beta(\mathsf{star}(R)) = \{\ & \langle \mathsf{set}(\epsilon), \mathsf{set}(\epsilon) \rangle\ \} \cup \\ & \{\ \langle \mathsf{rcon}(\mathsf{star}(R), r_1), \mathsf{rcon}(r_2, \mathsf{star}(R)) \rangle \mid \langle r_1, r_2 \rangle \in \beta(R)\ \}\end{aligned} \tag{4.66}$$

Figure 4.8. Beta function for concrete regular expressions, where c_i's are characters, and R_i's are regular expressions.

The splitting function correctly partitions a regular language with all possible cases, as shown in Lemma 34.

Lemma 34 For any string literals l_s and l_t, and for any concrete term R of sort Lan, if $l_s \cdot l_t$ in the language of R, then there exists a pair $\langle R_1, R_2 \rangle$ in $\beta(R)$, s.t. $l_s \in \mathcal{L}(R_1)$ and $l_t \in \mathcal{L}(R_2)$.

Proof: We prove by induction on the structure of a regular expression, following Formulas 4.61— 4.66 of the function β in Figure 4.8 for splitting a regular expression. Note that because the number of symbols in a regular expression is finite and the function β never introduces new symbols, it always terminates.

Base cases are Formulas 4.61— 4.62. By the definition, Formulas 4.61— 4.62 hold trivially.

For Formula 4.63, we prove by induction on the size of concatenation N (the number of regular expressions in the concatenation). The base case ($N = 0$ and $N = 1$) holds trivially. Now assume for all regular expressions with up to N concatenated expressions, the lemma holds. And assume we have $N + 1$ regular expressions. We pick one sub-regular expression in R, say R_i. By assumption, $\beta(R_i)$ splits R_i correctly and generates all possible pairs. Also, $\mathsf{rcon}(R_1, \cdots, R_n) \approx \mathsf{rcon}(\mathsf{rcon}(R_1, \cdots, R_{i-1}, r_1), \mathsf{rcon}(r_2, R_{i+1}, \cdots, R_n))$, for all r_1, r_2 pairs. Because for each pair of r_1 and r_2, $R_i \approx \mathsf{rcon}(r_1, r_2)$ and those are all splits for R_i by IH, the algorithm will return all possible splits.

Formula 4.64 holds because of the semantics of the operation union. Formula 4.65 holds because of Theorem 8. Formula 4.66 holds since $\mathsf{star}(R)$ is equal to $\mathsf{union}(\mathsf{set}(\epsilon), \mathsf{rcon}(R, \mathsf{star}(R)))$ and by a similar reason for Formula 4.63. ■

Concrete Length Function

We now extend the signature of the theory of regular expressions with one extra symbol $\mathsf{rep} : \mathsf{Lan} \times \mathsf{Int} \rightarrow \mathsf{Lan}$. rep is an operation that takes two terms: a term of sort Lan and a term of sort Int. The meaning of $\mathsf{rep}(R, i)$ is to represent i-replications of R in a concatenation, i.e., for any model $\mathcal{M}$, s.t.

$$\mathcal{M}[\![\mathsf{rep}(R, i)]\!] \Longleftrightarrow \mathcal{M}[\![\mathsf{rcon}(\underbrace{R, R, \cdots, R}_{\mathcal{M}[\![i]\!] \text{ copies}})]\!]. \tag{4.67}$$

Note that for an arbitrary application of the star operation, for an arbitrary string in $\mathcal{L}(\mathsf{star}(R))$, there exists a non-negative number n, s.t. this string is also in $\mathcal{L}(\mathsf{rep}(R, n))$. As a consequence, $\mathsf{in}(s, \mathsf{star}(R))$ is equivalent to $\exists i : \mathsf{Int}.\mathsf{in}(s, \mathsf{rep}(R, i))$ and $i \geq 0$ in the extended theory.

We call a regular expression is an extended regular expression if it contains symbols from the core signature as well as the symbol rep. If R is an extended regular expression and S is a set of linear arithmetic constraints, we use $\natural|R, S|\natural$ to denote a function that returns a length term and a set of linear arithmetic constraints, s.t. a length term (together with the set) that represents lengths of all strings that are in the language of R (if S is the empty set).

Our algorithm is described by the rules in Figure 4.9. It works from top down. Initially, we have a regular expression with the empty length condition: $\natural|R, S|\natural$, where S is a set of linear constraints over lengths and initially, it is the empty set.

Now we show that terms generated by the rules in Figure 4.9 are linear.

Lemma 35 For any regular expression R, if $\natural|R, \emptyset|\natural = \langle k, S\rangle$, both k and S contain

$$\text{¦|}\mathsf{set}(s), S\text{|¦} = \langle \mathsf{len}(s), S \rangle \tag{4.68}$$

$$\text{¦|}\mathsf{rcon}(R_1, R_2), S\text{|¦} = \langle k, S_1 \cup S_2 \cup \{k \approx k_1 + k_2\} \rangle \quad \text{if } \text{¦|}R_1, S\text{|¦} = \langle k_1, S_1 \rangle \text{ and } \text{¦|}R_2, S\text{|¦} = \langle k_2, S_2 \rangle \tag{4.69}$$

$$\text{¦|}\mathsf{star}(R), S\text{|¦} = \text{¦|}\mathsf{rep}(R, x), S \cup \{x \geq 0\}\text{|¦} \tag{4.70}$$

$$\text{¦|}\mathsf{rep}(\mathsf{set}(s), x), S\text{|¦} = \langle (x \times \mathsf{len}(s)), S \rangle \tag{4.71}$$

$$\text{¦|}\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x), S\text{|¦} = \langle k, S_1 \cup S_2 \cup \{k \approx k_1 + k_2\} \rangle \quad \text{if } \text{¦|}\mathsf{rep}(R_1, x), S\text{|¦} = \langle k_1, S_1 \rangle \text{ and } \text{¦|}\mathsf{rep}(R_2, x), S\text{|¦} = \langle k_2, S_2 \rangle \tag{4.72}$$

$$\text{¦|}\mathsf{rep}(\mathsf{union}(R_1, R_2), x), S\text{|¦} = \langle k, S_1 \cup S_2 \cup \{k \approx k_1 + k_2, x \approx x_1 + x_2, x_1 \geq 0, x_2 \geq 0\} \rangle \quad \text{if } \text{¦|}\mathsf{rep}(R_1, x_1), S\text{|¦} = \langle k_1, S_1 \rangle \text{ and } \text{¦|}\mathsf{rep}(R_2, x_2), S\text{|¦} = \langle k_2, S_2 \rangle \tag{4.73}$$

$$\text{¦|}\mathsf{rep}(\mathsf{star}(R), x_1), S\text{|¦} = \text{¦|}\mathsf{rep}(R, x_2), S \cup \{x_1 \approx 0 \Rightarrow x_2 \approx 0, x_2 \geq 0\}\text{|¦} \tag{4.74}$$

Figure 4.9. Concrete regular expression length rules, where s is a term of sort String, and k_i's, x_i's are non-negative terms of sort Int.

only linear arithmetic terms.

Proof: The only multiplication term is generated by Formula 4.71. Other formulas do not generate non-linearity. Because there is no variable in regular expressions, $\mathsf{len}(s)$ will return a natural number, and then $x \times \mathsf{len}(s)$ is still a linear term.

Thus, the property holds. ■

Lemma 36 For any regular expression R, the function for computing $\text{¦|}R, \emptyset\text{|¦}$, with respect to the rules in Figure 4.9, terminates.

Proof: Our termination measure is a lexicographic combination of :

1. the number of star operators,

2. the number of other regular expression operators.

If Formula 4.70 is applied, the first number decreases. In all other cases, the second number decreases.

Thus, the algorithm is terminating. ■

To simplify our proofs, we say a length n can be *produced* by a regular expression if there exists a string (in the language of that regular expression) whose length is n.

Lemma 37 For any string literal s, and for any regular expression R, s.t. $s \in \mathcal{L}(R)$ iff $\mathcal{M} \models \mathsf{len}(s) \approx k, S$, for some model $\mathcal{M}$, where $\natural|R, \emptyset|\natural = \langle k, S \rangle$.

Proof: Let R be an arbitrary regular expression. We show that $\natural|R, \emptyset|\natural$ returns a linear term (together with the set of current constraints) that represents all possible lengths of the strings that are in the language of R.

The proof is by induction on the number of steps to the termination, which also refers the final linear arithmetic term.

Base Case: If it is one step to the final length term, the regular expression has to be either $set(s)$ or $\mathsf{rep}(\mathsf{set}(s), x)$, since Formula 4.68 and 4.71 are the only ones generating linear arithmetic terms. If it is $\mathsf{set}(s)$, by Formula 4.68, the length term is $\mathsf{len}(s)$ and it is the only length that this regular expression can produce. If it is $\mathsf{rep}(\mathsf{set}(s), x)$, by definition, we know that the length term is $x \times \mathsf{len}(s)$, which reflects Formula 4.71.

Induction Hypothesis: If $\lVert R \rVert.\emptyset$ is n (or less) steps to a final term, the final term (together with the constraint set) represents all possible lengths that R can generate.

Step Case: Assume $\lVert R \rVert.\emptyset$ is $n+1$ steps to a final term. We consider all cases for R.

If R is of the form $\mathsf{rcon}(R_1, R_2)$, then Formula 4.69 is applicable. By definition of regular expressions, $\mathcal{L}(\mathsf{rcon}(R_1, R_2)) = \{s \cdot t \mid s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\}$. If k_1 represents an arbitrary length that R_1 can produce k_2 represents an arbitrary lengths that R_2 can produce, k represents a length that $\mathsf{rcon}(R_1, R_2)$ can produce, then k is $k_1 + k_2$.

Formula 4.70 is correct by definition of the extended symbol rep.

If R is of the form $\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x)$, then Formula 4.72 is applicable. By definition of regular expressions, the length $\mathcal{L}(\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x)) = \{(s \cdot t)^x \mid s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\}$. With respect to length only, this set is equal to $\{s^x \cdot t^x \mid s \in \mathcal{L}(R_1), t \in \mathcal{L}(R_2)\}$, or $\mathcal{L}(\mathsf{rep}(R_1, x)) \cup \mathcal{L}(\mathsf{rep}(R_2, x))$ If k_1 represents an arbitrary length that $\mathcal{L}(\mathsf{rep}(R_1, x))$ can produce, k_2 represents an arbitrary length that $\mathcal{L}(\mathsf{rep}(R_2, x))$ can produce, k represents a length that $\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x)$ can produce, then k is $k_1 + k_2$.

If R is of the form $\mathsf{rep}(\mathsf{union}(R_1, R_2), x)$, then Formula 4.73 is applicable. By definition of regular expressions, an arbitrary length of $\mathcal{L}(\mathsf{rep}(\mathsf{union}(R_1, R_2), x))$ can be represented by a length of $\mathcal{L}(\mathsf{rep}(R_1, x_1))$ plus a length of $\mathcal{L}(\mathsf{rep}(R_2, x_2))$, where x_1 and x_2 are non-negative integers and $x = x_1 + x_2$. Intuitively, if a string is in the language of x-fold of $\mathsf{union}(R_1, R_2)$, there exists two non-negative integer x_1 and x_2

and $x = x_1 + x_2$, and some part of the string must be in the language of x_1-fold of R_1 while the rest must be in the language of x_2-fold of R_2. Thus, if k_1 represents an arbitrary length that $\mathcal{L}(\mathsf{rep}(R_1, x_1))$ can produce, k_2 represents an arbitrary length that $\mathcal{L}(\mathsf{rep}(R_2, x_2))$ can produce, k represents a length that $\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x)$ can produce, then k is $k_1 + k_2$.

If R is of the form $\mathsf{rep}(\mathsf{star}(R), x_1)$, then Formula 4.74 is applicable. We consider two cases: either $x_1 = 0$ or $x_1 > 0$.

- Case $x_1 = 0$: It is clear that the language is a singleton set which only contains ϵ. Thus, it is equivalent to $\mathsf{rep}(R, 0)$. By Formula 4.71 in the next step, the only length produced by the regular expression is 0.

- Case $x_1 > 0$: If x_1 is not equal to 0, for an arbitrary string in the language of $\mathsf{rep}(\mathsf{star}(R), x_1)$, there exists another non-negative integer x_2 s.t. the string is also in the language of $\mathsf{rep}(\mathsf{star}(R), x_2)$.

Therefore, the algorithm preserves all possible lengths that a regular expression can generate. ■

4.1.4 Derivation Rules

In this subsection, we introduce our derivation rules for handling concrete membership constraints. These rules can be classified into the following groups: interaction, intersection, main and length reduction. These rules may depend on the functions in the previous subsection. We will show that our proof procedure is a decision procedure for concrete membership and length constraints.

$$\textbf{R-A-Prop}\quad \frac{\mathsf{len}(s) \approx 1 \in \mathsf{R}}{\mathsf{A} := \mathsf{A}, \mathsf{len}(s)\downarrow \approx 1}$$

$$\textbf{R-S-Prop}\quad \frac{s \approx t \in \mathsf{R}}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\textbf{R-Conflict}\quad \frac{\mathsf{false} \in \mathsf{R}}{\mathsf{unsat}}$$

Figure 4.10. Rules for interactions, where s and t are terms of String.

Interaction Rules

The rules in Figure 4.10 model the interactions among R, S, and A. By the normalization rules, some membership constraints will be normalized to the form $\mathsf{len}(s) \approx 1$. This equality will be passed to A by the rule **R-A-Prop**. Similarly, if a membership constraint is normalized to $s \approx t$, where one of the two terms is a string literal, this equality will be passed to S by the rule **R-S-Prop**. If a membership constraint is normalized to false, the derivation is closed by the rule **R-Conflict**.

Intersection Rules

We introduce two intersection rules in Figure 4.11. In **R-C-Inter-1**, the procedure gets a concrete regular expression whose top symbol is inter. The procedure tries to replace the intersection with a new regular expression by the intersection algorithm. Similarly, when the same string is in two separate languages (recognized by different regular expressions), we try to merge these two regular expressions by the intersection algorithm in **R-C-Inter-2**.

Notice that the correctness is proved by Lemma 32. After applying either **R-C-Inter-1** or **R-C-Inter-2**, either the number of unprocessed regular membership

$$\textbf{R-C-Inter-1} \quad \frac{\mathsf{in}(s, \mathsf{inter}(R_1, R_2)) \in \mathsf{R} \quad \mathcal{V}(R_1) = \mathcal{V}(R_2) = \emptyset}{\mathsf{R} := \mathsf{R}, \mathsf{in}(s, \pi(R_1, R_2))\downarrow}$$

$$\textbf{R-C-Inter-2} \quad \frac{\mathsf{in}(s, R_1), \mathsf{in}(t, R_2) \in \mathsf{R} \quad \mathsf{N}[s] = \mathsf{N}[t] \quad \mathcal{V}(R_1) = \mathcal{V}(R_2) = \emptyset}{\mathsf{R} := \mathsf{R}, \mathsf{in}(s, \pi(R_1, R_2))\downarrow}$$

Figure 4.11. Rules for handling the intersection of two concrete regular expressions, where s and t are terms of String, and R_1, R_2 are regular expressions.

constraints or the number of intersections decreases. Since those numbers are finite at the beginning, the number of applications of these two rules is finite.

Main Derivation Rules

Now, we introduce our main deduction rules in Figure 4.12. Note that regular expressions contain no string variables, and after rewriting strings on the left hand side of membership constrains are either a variable or a concatenation of *atomic strings*.

If a string is a concatenation, it has to be begun with either a character or a variable. If a string starts with a character, the rule **R-Consume** can be applied and it consumes this character; If a string starts with a variable, the rule **R-Split** can be applied and in each branch we get two more membership constraints, where one restricts the variable and the other restricts the rest of the string.

Lemma 38 If a closed derivation tree with an initial configuration $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ and the derivation involves applications of all rules in Chapter 3 as well as **R-Consume** and **R-Split**, then $S_0 \cup A_0 \cup R_0$ is unsatisfiable.

Proof: This lemma can be viewed as an extension of Theorem 1, where additional

term rewriting rules and **R-Consume** and **R-Split** are added to the calculus. Because of Lemmas 28, 8 and 24, we only need to focus on the correctness of **R-Consume** and **R-Split**. In other words, it is enough to show that the premises are equivalent to the conclusion in either rule.

For **R-Consume**, $\mathsf{in}(\mathsf{con}(c, s), R)$ is equal to $\mathsf{in}(s, \partial_c(R))$ by Theorem 7. For **R-Split**, $\mathsf{in}(\mathsf{con}(x, s), R)$ is equal to $\bigcup_{\langle r_1, r_2\rangle \in \beta(R)}(\mathsf{in}(x, r_1) \wedge \mathsf{in}(s, r_2))$ by Lemma 34.

Thus, the property holds. ■

Note that applying the same rule to the same membership constraint will not generate new constraints. We call a membership constraint *processed* if either **R-Consume** or **R-Split** has applied to that constraint to completion; otherwise, it is *unprocessed*. Also, **R-Consume** and **R-Split** cannot apply simultaneously to the same constraint because the premises of these two rules are disjoint.

Because the number of symbols in the left hand side of a membership constraint is finite and every time when either **R-Consume** or **R-Split** that number decreases, the derivation will terminate either being closing or being saturated by either **R-Consume** or **R-Split**.

Lemma 39 If a derivation tree with an initial configuration $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ is saturated with configuration $\langle S_n, A_n, R_n, F_n, N_n, C_n, B_n\rangle$ (by all rules up to now expect **R-Star**), then either all constraints in R_n are processed or unprocessed constraints in R_n are in the form of $\mathsf{in}(x_i, R_i)$, where each x_i appears exactly once in the unprocessed set.

Proof: For an arbitrary membership constraint $\mathsf{in}(s, R)$, the string term s must be

in one of the forms: (i) a string constant (e.g., l), (ii) a variable of sort String (e.g., x), (iii) a string term starting with a character (e.g., $\mathsf{con}(c, t)$), or (iv) a string term starting with a variable (e.g., $\mathsf{con}(x, t)$).

Because the derivation tree is *saturated* by all rules up to now expect **R-Star**,

- An unprocessed constraint cannot be in the form (i); otherwise, a rewriting rule can be applied.
- An unprocessed constraint cannot be in the form (iii); otherwise, the rule **R-Consume** can be applied.
- An unprocessed constraint cannot be in the form (iv); otherwise, the rule **R-Split** can be applied.

Thus, the only unprocessed ones are in the form (ii).

If a configuration is saturated by all rules (up to now) expect **R-Star**, we will not have two unprocessed constraints in the form (ii) with the same string variable. Assume that we are left with two unprocessed constraints in the form (ii) with the same string variable (i.e., $\mathsf{in}(x, R_1)$ and $\mathsf{in}(x, R_2)$), the rule **R-C-Inter** will be applied.

Therefore, after saturation by all rules (up to now) expect **R-Star**, either we have no unprocessed constraints in R_n, or constraints in the unprocessed set in R_n are $\mathsf{in}(x_1, R_1), \cdots, \mathsf{in}(x_m, R_m)$, where all x_i's are distinct. ■

By Lemma 39, now we only need to consider solving constraints of the form $\mathsf{in}(x, R)$ and length constraints.

$$\textbf{R-Consume} \quad \frac{\mathsf{in}(t, R) \in \mathsf{R} \quad \mathsf{N}\,[t] = (c, \boldsymbol{u})}{\mathsf{R} := \mathsf{R}, \mathsf{in}(\mathsf{con}(\boldsymbol{u}), \partial_c(R))\!\downarrow}$$

$$\textbf{R-Split} \quad \frac{\mathsf{in}(t, R) \in \mathsf{R} \quad \mathsf{N}\,[t] = (x, \boldsymbol{u})}{\|_{\langle R_1, R_2\rangle \in \beta(R)} \;\; \mathsf{R} := \mathsf{R}, \mathsf{in}(x, R_1)\!\downarrow, \mathsf{in}(\mathsf{con}(\boldsymbol{u}), R_2)\!\downarrow}$$

Figure 4.12. Concrete membership derivation rules, where t is a term of String, and R, R_1, R_2 are regular expressions.

A length constraint of a regular expression provides all possible lengths of strings that are in the language of this regular expression.

Length Rules

One way to solve this problem is to add length constraints for such membership constraints. However, the naïve method may introduce some non-linear constraints which may not be solved.

Our approach is to generate, for the regular expression in each unprocessed constraint, a set of linear arithmetic constraints that represents all possible lengths of strings in the language of this regular expression. Then, we send linear arithmetic constraints to the LIA solver.

Before discussing our algorithm, we introduce one more set of reduction rules for simplifying unit membership constraints in Figure 4.13. Correctness follows the definition of regular expressions. After applying these rules, the unprocessed constraints are in the form of $\mathsf{in}(x, \mathsf{star}(R))$.

Now we introduce our regular expression rule for generating length constraints over membership constraints in Figure 4.14. Note that k is a linear term of sort Int and S is a set of additional linear constraints. They are computed by the algorithm

$$\textbf{R-S-Concat} \quad \frac{\mathsf{in}(x, \mathsf{rcon}(R_1, R_2)) \in \mathsf{R} \quad \mathsf{N}\,[x] = (x)}{\mathsf{R} := \mathsf{R}, \mathsf{in}(k_1, R_1)\!\downarrow, \mathsf{in}(k_2, R_2)\!\downarrow \quad \mathsf{S} := \mathsf{S}, x \approx \mathsf{con}(k_1, k_2)}$$

$$\textbf{R-S-Union} \quad \frac{\mathsf{in}(x, \mathsf{union}(R_1, R_2)) \in \mathsf{R} \quad \mathsf{N}\,[x] = (x)}{\mathsf{R} := \mathsf{R}, \mathsf{in}(x, R_1)\!\downarrow \quad \| \quad \mathsf{R} := \mathsf{R}, \mathsf{in}(x, R_2)\!\downarrow}$$

Figure 4.13. Unit membership reduction rules, where x is a term of String, and R_1, R_2 are regular expressions.

$$\textbf{R-Len} \quad \frac{\mathsf{in}(x, R) \in \mathsf{R} \quad \natural|R, \emptyset|\natural = \langle k, S\rangle}{\mathsf{A} := \mathsf{A}, \mathsf{len}(x) \approx k\!\downarrow, S\!\downarrow}$$

Figure 4.14. Regular membership length rule, where k is a term of Int, and S is a set of linear arithmetic constraints.

described in Figure 4.9.

Linear arithmetic constraints generated by **R-Len** will be sent to the LIA engine. Note that before applying this rule, we only have membership constraints of the form $\mathsf{in}(x, R)$ in R. In other words, if we assign any string in R to x, R will be satisfied. If there is no additional arithmetic constraint from the input, those membership constraints can be trivially satisfied (e.g., simply by assigning the first enumerated string in the language of R). If there are some additional arithmetic constraints from the input, the LIA engine will check the consistence with the ones added by **R-Len**. If the LIA engine finds a conflict, then this branch will be closed; otherwise, the derivation tree is saturated.

4.1.5 Correctness

We now show that our calculus for handling regular membership constraints together with word equations and length constraints is *Refutation Soundness* and

Solution Soundness. We use T_{SRL} to denote the theory of unbounded strings, regular membership and length constraints.

Theorem 9 (Refutation Soundness for constraints over T_{SRL}) *For any closed derivation tree rooted by an initial configuration $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, the set $S_0 \cup A_0 \cup R_0$ is unsatisfiable in T_{SRL}.*

Proof: The proof is as an extension of Lemma 38, which is an extension of Theorem 1. Our hypothesis is that for all closed derivation trees with depth of at most n and an initial configuration $\langle S, A, R, F, N, C, B\rangle$, the set $S \cup A \cup R$ is unsatisfiable in T_{SRL}. Thus, we only need to show refutation soundness in the induction step when **R-Len** is applied.

Let $c_0 = \langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ be the configuration of the root of a closed derivation tree of depth $n+1$, the rule **R-Len** is applied at the first step, we want to show that the set $S_0 \cup A_0 \cup R_0$ is unsatisfiable in T_{SRL}.

Assume $c_1 = \langle S_1, A_1, R_1, \cdots\rangle$ is the configuration of the child node after applying **R-Len** to some membership constraint $\mathsf{in}(x, R)$, and $\natural|R, \emptyset|\natural = \langle k, S\rangle$. By hypothesis, we know that $S_1 \cup A_1 \cup R_1$ is unsatisfiable. By Lemma 37, we have $\mathsf{in}(x, R) \models_{T_{\mathsf{SRL}}} \mathsf{len}(x) \approx k,\ S'$, where $\natural|R, \emptyset|\natural = \langle k, S'\rangle$. By **R-Len**, we know $S_0 = S_1$, $R_0 = R_1$ and $A_1 = A_0 \cup \{\mathsf{len}(x) \approx k\} \cup S'$, where $\natural|R, \emptyset|\natural = \langle k, S'\rangle$. Thus, $S_0 \cup A_0 \cup R_0$ is unsatisfiable.

This concludes our proof. ■

To prove the solution soundness, we introduce some more definitions. An extended regular expression is called *flattened* if it is in the form of

$$\mathsf{rcon}(R_1, \cdots, R_n), \tag{4.75}$$

where R_i is either an *atomic regular expression* (i.e., rempty or $\mathsf{set}(s)$) or a rep term of an *atomic regular expression* (e.g., $\mathsf{rep}(\mathsf{set}(s), n)$).

Let R be an arbitrary regular expression. Consider rules in Figure 4.14 expect Formulas 4.68 and 4.71 and ignore the set S. If we apply the length function to R to completion, we will eventually get a new regular expression, which is one step to the final arithmetic term if we apply all these rules. We call this regular expression the *flattened form* of R. The *flattened form* of R is a flattened regular expression.

Lemma 40 Let $\mathsf{in}(t, R)$ be an arbitrary membership constraint, $\mathsf{rcon}(\mathsf{rep}(\mathsf{set}(s_1), x_1), \cdots, \mathsf{rep}(\mathsf{set}(s_m), x_m))$ be the flattened form of R (generated by rules in Figure 4.14 expect Formulas 4.68 and 4.71), where x_i's are natural numbers. Then, $s_1^{x_1} \cdots s_m^{x_m}$ is a string in $\mathcal{L}(R)$.

Proof: We show the proof by induction of the number of steps that we use to build the flatten form of a regular expression based on the rules in Figure 4.9.

Base Case: If a regular expression is in the form of either $\mathsf{set}(s)$ or $\mathsf{rep}(\mathsf{set}(s), x)$, the flatten form is itself. s is the only string in the language of $\mathsf{set}(s)$; s^x is the only string in the language of $\mathsf{rep}(\mathsf{set}(s), x)$, where $n \geq 0$.

Induction Hypothesis (IH): If a regular expression R need n steps (or less) to build its flatten form ($\mathsf{rcon}(\mathsf{rep}(\mathsf{set}(s_1), x_1), \cdots, \mathsf{rep}(\mathsf{set}(s_m), x_m))$) based on the rules

in Figure 4.9, then $s_1^{x_1} \cdots s_m^{x_m}$ is a string in $\mathcal{L}(R)$.

Step Case: Assume a regular expression R need $n+1$ steps. We show all cases.

If a regular expression is in the form of $\mathsf{rcon}(R_1, R_2)$, by IH, we have r_1 is the flattened form of R_1 and r_2 is the flattened form of R_2. By definition, we know for any string s in $\mathcal{L}(r_1)$ is in $\mathcal{L}(R_1)$ and any string t in $\mathcal{L}(r_2)$ is in $\mathcal{L}(R_2)$, $s \cdot t$ is in $\mathcal{L}(\mathsf{rcon}(R_1, R_2))$.

If a regular expression is in the form of $\mathsf{star}(R)$, by IH we have r is the flattened form of $\mathsf{rep}(R, n)$. By definition, we know for any string s in $\mathcal{L}(r)$ is in $\mathcal{L}(\mathsf{rep}(R, n))$. s is also in $\mathcal{L}(\mathsf{star}(R))$.

If a regular expression is in the form of $\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x)$, by IH we have r_1 is the flattened form of $\mathsf{rep}(R_1, x)$ and r_2 is the flattened form of $\mathsf{rep}(R_2, x)$. By definition, we know for any string s in $\mathcal{L}(r_1)$ is in $\mathcal{L}(\mathsf{rep}(R_1, x))$ and any string t in $\mathcal{L}(r_2)$ is in $\mathcal{L}(\mathsf{rep}(R_2, x))$, $\mathsf{con}(s, t)$ is in $\mathcal{L}(\mathsf{rep}(\mathsf{rcon}(R_1, R_2), x))$.

If a regular expression is in the form of $\mathsf{rep}(\mathsf{union}(R_1, R_2), x)$, by IH we have r_1 is the flattened form of $\mathsf{rep}(R_1, x_1)$ and r_2 is the flattened form of $\mathsf{rep}(R_2, x_2)$. By definition, we know for any string s in $\mathcal{L}(r_1)$ is in $\mathcal{L}(\mathsf{rep}(R_1, x_1))$, and any string t in $\mathcal{L}(r_2)$ is in $\mathcal{L}(\mathsf{rep}(R_2, x_2))$. $s \cdot t$ is in $\mathcal{L}(\mathsf{rep}(\mathsf{union}(R_1, R_2), x))$.

If a regular expression is in the form of $\mathsf{rep}(\mathsf{star}(R), x_1)$, by IH we have r is the flattened form of $\mathsf{rep}(R, x_2)$. By definition, we know for any string s in $\mathcal{L}(r)$ is in $\mathcal{L}(\mathsf{rep}(R, x_2))$, s is also in $\mathcal{L}(\mathsf{rep}(\mathsf{star}(R), x_1))$.

Therefore, the property holds.

By Lemma 40, if we have $\mathsf{in}(x, R)$, we can build a model for x from the flattened form of R.

Example 11 The flattened form of $\mathsf{star}(\mathsf{rcon}(\mathsf{star}(\mathsf{set}(aa)), \mathsf{set}(bbb), \mathsf{star}(\mathsf{set}(aaa))))$ is $\mathsf{rcon}(\mathsf{rep}(\mathsf{set}(aa), x_1), \mathsf{rep}(\mathsf{set}(bbb), x_2), \mathsf{rep}(\mathsf{set}(aaa), x_3))$. Assume we have a membership constraint $\mathsf{in}(x, \mathsf{star}(\mathsf{rcon}(\mathsf{star}(\mathsf{set}(aa)), \mathsf{set}(bbb), \mathsf{star}(\mathsf{set}(aaa)))))$. We add the regular expression length constraints for x, if the arithmetic engine finds a model for x_1, x_2, x_3, say $x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3$, we can build a model for x based on the arithmetic model, which is $x \mapsto aabbbbbbaaaaaaaaa$. □

Lemma 41 Let $\mathsf{in}(x, R)$ be the only unprocessed membership constraint in R. If its length constraints are added by **R-Len** and a LIA engine finds a model, we can always build a model for x.

Proof: Note that we always process membership constraints after processing word equations. Because of rules in Figure 4.13, the regular expression is in the form of $\mathsf{star}(R_1)$. If there is no additional length constraint, any string in the language of $\mathsf{star}(R_1)$ is a model for x, thus, we can simply assign x to the empty string. If there are some length constraints on x, we need to find a string (in the language of $\mathsf{star}(R_1)$) that satisfies length constraints.

By Lemma 37, we know that the added length term represents all possible lengths that the regular expression can generate. If there is no conflict in LIA, we know that we can find a length for x that is compliant with both regular expression and length constraints.

By Lemma 40, we can construct a model for x based on the models of coefficients in the flattened form of R. ■

Theorem 10 (Solution Soundness for constraints over T_{SRL}) *If a derivation tree with an initial configuration $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ contains a saturated configuration, then the set $S_0 \cup A_0 \cup R_0$ is satisfiable in T_{SRL}. The saturated configuration induces a satisfying assignment for the set $S_0 \cup A_0 \cup R_0$ in T_{SRL}.*

Proof: The proof is an extension of Theorem 2. We need to focus on how to build a model for variables: (i) their normal forms are themselves, and (ii) they are in the membership constraints of the form $\mathsf{in}(x, R)$.

Note that because the derivation tree is saturated, we have the length for x and also all the assignments for coefficients of the *flattened form* of R. By Lemma 41, we can construct a model for x, and also the model of x is indeed a model.

Therefore, our calculus is solution sound. ■

4.1.6 Decision Procedure

In this subsection, we provide the proofs of both refutation completeness and solution completeness for our calculus over membership and length constraints. In addition, we prove that our approach is a decision procedure for T_{RL}.

Theorem 11 (Termination for constraints over T_{RL}) *For any derivation tree rooted by an initial configuration $\langle \emptyset, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, the derivation tree is finite.*

Proof: By Lemma 39, we know that given an initial configuration $\langle \emptyset, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, it terminates either with an unsat result, or with a saturated configuration $\langle S_i, A_i, R_i,$

$F_i, N_i, C_i, B_i\rangle$ (without regular expression length rules). The only unprocessed constraints in R_i are in the form of $\mathsf{in}(x_i, R_i)$. Moreover, because of rules in Figure 4.13, all R_i are in the form of $\mathsf{star}(R_i')$. In the first case where the derivation is closed, the termination is obvious. Thus, we only need to show the termination for the second case where the derivation tree is saturated without the rule **R-Len**.

Assume an arbitrary derivation subtree with a configuration $\langle S_0, A_0, R_0, F_0, N_0, C_0, B_0\rangle$, where constraints in R_0 are in the form of $\mathsf{in}(x_i, r_i)$, $N_0\,[x_i] = (x_i), \forall x_i$, and x_i's are distinct.

Note that the only possibly applicable rule is **R-Len**. By Lemma 37, all added constraints are entailed by R_0 in T_{RL}. There is no change for S or R. All constraints (in A) are linear arithmetic constraints from **R-Len**. By assumption of the LIA behavior, the LIA engine will terminate. Since S will not be changed after the termination of the LIA engine, the procedure terminates.

Therefore, the termination property holds. ■

Theorem 12 (Solution Completeness for constraints over T_{RL}) *For any derivation tree rooted by an initial configuration $\langle \emptyset, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, where the set $A_0 \cup R_0$ is satisfiable in T_{RL}, the derivation tree is saturated.*

Proof: By Theorem 11, for any arbitrary configuration, the derivation terminates. if the set $A_0 \cup R_0$ is satisfiable in T_{RL}, the derivation trees will be terminating. It terminates either with a closed derivation tree or with a saturated derivation tree.

By Theorem 9, if it terminates with a closed derivation tree, $A_0 \cup R_0$ is unsatisfiable. By Theorem 10, if it terminates with a saturated derivation tree, $A_0 \cup R_0$ is

satisfiable.

In other words, if $A_0 \cup R_0$ is satisfiable, and the derivation tree is closed, Theorem 9 would be violated.

Therefore, our calculus is solution complete. ■

Theorem 13 (Refutation Completeness for constraints over T_{RL}) *For any derivation tree rooted by an initial configuration $\langle \emptyset, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, where the set $A_0 \cup R_0$ is unsatisfiable in T_{RL}, the derivation tree is closed.*

Proof: Because our calculus is solution complete by Theorem 12, it is also refutation sound by Theorem 9, and the derivation terminates for every configuration by Theorem 11, it is refutation complete. ■

Theorem 14 (Decision Procedure for constraints over T_{RL}) *Let t be a derivation tree with an initial configuration $\langle \emptyset, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. If $A_0 \cup R_0$ is satisfiable in T_{RL}, the derivation tree is saturated; otherwise, it is closed.*

Proof: Since our calculus is refutation complete (by Theorem 13), to prove our calculus is a decision procedure, it is enough to show that for all satisfiable problems, the derivation trees will be saturated, and for all unsatisfiable problems, the derivation trees will be closed.

Since our calculus is refutation complete by Theorem 13, refutation sound by Theorem 9 and terminating by Theorem 11, it is a decision procedure for T_{RL}. ■

4.2 Extensions

In this section, we will discuss two extensions of our calculus with respect to membership constraints. We introduce additional rules for solving symbolic membership constraints and for handling negative membership constraints lazily.

4.2.1 Symbolic Language Extension

Rules for handling symbolic membership constraints are extension to our calculus. Since rules in Figure 4.12 are the main rules in our calculus, these rules depend on the derivative function and the beta function. We expand the definition of these functions first.

Note that if the language is denoted by concrete regular expression, we refer to the rules in the previous section.

Symbolic Delta Function

The expanded symbolic delta function δ, takes a symbolic regular expression as an input and returns a logical formula. The logical formula is evaluated to be true iff the symbolic regular expression contains the empty string. The rules are shown in Figure 4.15. Compared to rules for the constant delta function (shown in Figure 4.4), the major differences are

- The result of constant delta function is a Boolean value, while the result of symbolic delta function is a logical formula.
- Formulas 4.27 and 4.28 are replaced by Formula 4.77. Note that s in Formula 4.77 may contain variables of sort String.

$$\delta(\mathsf{rempty}) = \mathsf{false} \tag{4.76}$$

$$\delta(\mathsf{set}(s)) = s \approx \epsilon \tag{4.77}$$

$$\delta(\mathsf{rcon}(R_1, \cdots, R_n)) = \delta(R_1) \wedge \cdots \wedge \delta(R_n) \tag{4.78}$$

$$\delta(\mathsf{union}(R_1, \cdots, R_n)) = \delta(R_1) \vee \cdots \vee \delta(R_n) \tag{4.79}$$

$$\delta(\mathsf{inter}(R_1, \cdots, R_n)) = \delta(R_1) \wedge \cdots \wedge \delta(R_n) \tag{4.80}$$

$$\delta(\mathsf{star}(R)) = \mathsf{true} \tag{4.81}$$

Figure 4.15. Delta function for symbolic regular expressions, where s is a term of String, and R_1 and R_2 are regular expressions.

- Formulas 4.29— 4.31 are applied eagerly, while Formulas 4.78— 4.80 are applied lazily due to the possibility of variable existence.

Except for the above differences, the behavior is identical to its counterpart. Therefore, when context is clear, we override the function symbol.

Symbolic Derivative Function

We extend the derivative function for concrete regular expression in Figure 4.6. Because of the existence of string variables in a regular expression, it is hard to eagerly compute a regular expression that fits the definition of derivative. Thus, we introduce a new function $æ_{c,t}$ which mimics the behavior of the derivative in constant cases. It splits the regular expression lazily.

The function $æ_{c,t}$ is always associated with a single constant character c and a string term t, where c has a similar meaning as the one in the constant derivative

$$æ_{c,t}(\mathsf{rempty}) = \mathsf{false} \tag{4.82}$$

$$æ_{c,t}(\mathsf{set}(s)) = s \approx \mathsf{con}(c,t) \tag{4.83}$$

$$\begin{aligned} æ_{c,t}(\mathsf{rcon}(R_1,R_2)) = &(\delta(R_1) \wedge æ_{c,t}(R_2)) \vee \\ &(t \approx \mathsf{con}(k_1,k_2) \wedge æ_{c,k_1}(R_1) \wedge \mathsf{in}(k_2,R_2)) \end{aligned} \tag{4.84}$$

$$æ_{c,t}(\mathsf{union}(R_1,R_2)) = æ_{c,t}(R_1) \vee æ_{c,t}(R_2) \tag{4.85}$$

$$æ_{c,t}(\mathsf{inter}(R_1,R_2)) = æ_{c,t}(R_1) \wedge æ_{c,t}(R_2) \tag{4.86}$$

$$æ_{c,t}(\mathsf{star}(R)) = t \approx \mathsf{con}(k_1,k_2) \wedge æ_{c,k_1}(R) \wedge \mathsf{in}(k_2,\mathsf{star}(R)) \tag{4.87}$$

Figure 4.16. Derivative function for symbolic regular expressions, where s and t are terms of String, c is a character, and R, R_1, R_2 are regular expressions. k_1, k_2 are fresh variables of String.

function, t is a term of sort String that refers to a suffix of the string in a membership constraint (e.g., if we have $\mathsf{in}(\mathsf{con}(c,t),R)$, the rule **R-Consume** is applied, and it adds $æ_{c,t}R$ to G). This function takes a symbolic regular language R and returns a formula F, s.t. $\mathsf{in}(\mathsf{con}(c,t),R)$ is evaluated to be true iff F is evaluated to be true.

Extended Proof Procedure

We describe our proof procedure as an expansion of the one in Section 3.1. Our procedure depends on the procedure of solving word equations. This procedure happens after the procedure for solving word equations. Note that if a derivation is saturated with respect to all rules described in Section 3.1, all string variables have their normal forms.

Definition 19 Let $c = \langle \mathsf{S},\mathsf{A},\mathsf{R},\mathsf{F},\mathsf{N},\mathsf{C},\mathsf{B}\rangle$ be a configuration, and R is a regular

expression in some membership constraint in R. We call R' is a *semi-normal form* of a regular expression if R' is obtained by replacing each variable in R with its normal form, i.e., $R' = R[x_i \mapsto \mathsf{N}\,[x_i]], \forall x_i \in \mathcal{V}(R)$. □

We override the normalization function symbol for this operation: we use $R' = \mathsf{N}\,R$ to denote that R' is a *semi-normal form* of R based on the configuration c.

The main procedure is based on the repeated application of the calculus rules according to the following stages:

Stage 1: Normalize string variables in word equations. In this stage, we apply all rules of Chapter 3 to completion. If it returns unsat, the input constraints are unsat regardless of membership constraints; otherwise, it is saturated by rules for word equations.

Stage 2: Compute *semi-normal form* for each regular expression. Because the derivation tree is saturated by rules for word equations, we know that for every variable of sort String, we have its normal form in N. Thus, we can replace each regular expression by its semi-normal form. The rationale of this stage is that after semi-normalization we have a chance to *neutralize* some symbolic regular expressions (by converting them into concrete regular expression). Thus, we have some regular membership constraints and length constraints. Based on Theorem 14, we know that our calculus is a decision procedure for the theory of concrete regular membership and length constraints, and our procedure will terminate. Note that in

general, if we have symbolic membership constraints, our procedure is not guaranteed to be terminating.

Stage 3: Apply rules for (symbolic) regular expressions. After *semi-normalization* in the previous stage, if we still have the unprocessed membership constraints in R, we pick one unprocessed constraint. If the regular expression of the constraint is constant, we process it and we know that our calculus is a decision procedure for this theory; otherwise, we try to apply additional rules in Figure 4.17.

Note that the language of symbolic regular expressions can be a context-sensitive language.

Example 12 Consider the symbolic regular expression $\mathsf{rcon}(\mathsf{set}(x), \mathsf{set}(y), \mathsf{set}(z))$, where $\mathsf{in}(x, \mathsf{star}(\mathsf{set}(a)))$, $\mathsf{in}(y, \mathsf{star}(\mathsf{set}(b)))$, $\mathsf{in}(z, \mathsf{star}(\mathsf{set}(c)))$ and $\mathsf{len}(x) \approx \mathsf{len}(y) \approx \mathsf{len}(z) \not\approx 0$.

By pumping lemma, we can show that this language is not context-free. □

Although it is known that the emptiness problem for context-sensitive languages is undecidable [47], the decidability for the satisfiability problem of a symbolic regular language still remains open. Since the satisfiability problem is harder than the constant membership problem in general, it is unclear whether our calculus for a general symbolic regular language will terminate for every problem (mainly because of the rule **R-S-Star**).

The rule **R-Consume** is not applicable for symbolic membership constraints

$$\textbf{R-S-Consume} \quad \frac{\mathsf{in}(t, R) \in \mathsf{R} \quad \mathsf{N}[t] = (c, \boldsymbol{u}) \quad \mathsf{N}R = R' \quad \mathcal{V}(R') \neq \emptyset}{\mathsf{G} := \mathsf{G}, \partial_{c,t}(R')}$$

$$\textbf{R-S-Split} \quad \frac{\mathsf{in}(t, R) \in \mathsf{R} \quad \mathsf{N}[t] = (x, \boldsymbol{u}) \quad \mathsf{N}R = \mathsf{star}(R') \quad \mathcal{V}(R') \neq \emptyset}{\begin{array}{c}\mathsf{G} := \mathsf{G}, x \approx \mathsf{con}(k_1, k_2), \mathsf{con}(\boldsymbol{u})\downarrow\approx \mathsf{con}(k_3, k_4), \\ \mathsf{in}(k_1, \mathsf{star}(R')), \mathsf{in}(k_4, rstar(R')), \mathsf{in}(\mathsf{con}(k_2, k_3), R')\end{array}}$$

$$\textbf{R-S-Star} \quad \frac{\begin{array}{c}\mathsf{in}(\mathsf{con}(x, s), R) \in \mathsf{R} \quad \mathsf{N}R = \mathsf{star}(R') \\ \mathcal{V}(R)' \neq \emptyset \quad \mathcal{K}(\mathsf{S}) \models \mathsf{con}(x, s) \not\approx \epsilon\end{array}}{\mathsf{G} := \mathsf{G}, \mathsf{con}(x, s) \approx k_1 k_2, \mathsf{in}(k_1, R'), \mathsf{in}(k_2, \mathsf{star}(R'))}$$

Figure 4.17. Symbolic membership derivation rules, where s, t are terms of String, x is a string variable, c is a character, $\boldsymbol{u}$ is a vector of string terms, and k_1, k_2, k_3, k_4 are fresh string variables.

because the original derivative function only works on constant membership constraints. Thus, we introduce the rule **R-S-Consume**. Note that **R-S-Consume** works with our extended symbolic derivative function (shown in Figure 4.16). The major difference is that the symbolic derivative function generates a formula (instead of a regular expression). This allows us to split the string variable t lazily.

The rule **R-Split** is not also applicable for symbolic membership constraints because Formula 4.62 requires splitting a string constant. Instead, we introduce another rule (**R-S-Split**) which mimics the behavior of the rule **R-Split** for splitting symbolic regular expressions. Note that due to the rules in Figure 4.13, we always get membership constraints in the form of $\mathsf{in}(x, \mathsf{star}(R))$, and since R' is a symbolic regular expression, the termination is not guaranteed.

In addition, we do not apply the rule **R-Len** because the constraints generated by $|\!|R|\!|$ are not linear in general. For instance, if we have $\mathsf{in}(x, \mathsf{star}(\mathsf{set}(y)))$, one possible new length constraint is $\mathsf{len}(x) \approx \mathsf{len}(y) \times k$, where k is a fresh variable of

sort Int. In general, non-linear arithmetic is undecidable. Unless we have a proper way to handle this sort of non-linear constraints, we have non-termination in general.

When we cannot apply the rule **R-Len** and as a workaround, we introduce the rule **R-S-Star**. This rule is similar to the rule **R-Star** (which is disabled in this chapter); however, we limit its application only to symbolic membership constraints which have no other applicable rules in our calculus. This is also a reason why our calculus is not guaranteed to be terminating for all symbolic membership constraints.

4.2.2 Negative Membership Extension

Another extension is with respect to negative membership constraints. Although it is known that regular languages are *closed* under complementation, the complexity of complementation computation is extremely high.

A classical method to compute the complement of a regular expression takes the following steps:

1. it first computes an NFA which recognizes the language of the regular expression;

2. then, it converts the NFA to a DFA (in general the number of states in DFA grows exponentially);

3. it computes a new NFA that recognizes the complement of the language which is recognized by the old DFA (by flipping all accept states to intermediate states, and vice versa);

4. finally, it converts the NFA back into a regular expression.

Therefore, we need to avoid this sort of expensive operations as much as possible. One idea is to solve negative membership constraints without applying the complementation operation. We adopt the idea of finite model finding in solving quantified formulas [76]. We reduce negative membership constraints to quantified formulas over bounded integers. The algorithm is shown in Figure 4.18.

R-N-Empty
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{rempty})}{\mathsf{R} = \mathsf{R}'}$$

R-N-Sigma
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{allchars})}{\mathsf{R} = \mathsf{R}' \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(s) \not\approx 1}$$

R-N-Const
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{set}(t))}{\mathsf{R} = \mathsf{R}' \quad \mathsf{S} := \mathsf{S}, s \not\approx t}$$

R-N-Concat
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{rcon}(R_1, R_2))}{\begin{array}{c}\mathsf{R} := \mathsf{R}' \quad \mathsf{Q} := \mathsf{Q}, \forall i : \mathsf{Int}.\ 0 \leq i \leq \mathsf{len}(s) \implies \\ (\mathsf{substr}(s, 0, i) \notin R_1 \vee \mathsf{substr}(s, i, \mathsf{len}(s) - i) \notin R_2)\end{array}}$$

R-N-Union
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{union}(R_1, \cdots, R_n))}{\mathsf{R} = \mathsf{R}', s \notin R_1, \cdots, s \notin R_n}$$

R-N-Inter
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{inter}(R_1, \cdots, R_n))}{\mathsf{R} := \mathsf{R}', s \notin R_1 \parallel \quad \cdots \quad \parallel \mathsf{R} := \mathsf{R}', s \notin R_n}$$

R-N-Star
$$\frac{\mathsf{R} = \mathsf{R}', \neg\mathsf{in}(s, \mathsf{star}(R)) \quad \mathsf{S} \models s \not\approx \epsilon}{\begin{array}{c}\mathsf{R} := \mathsf{R}' \quad \mathsf{Q} := \mathsf{Q}, \forall i : \mathsf{Int}.\ 1 \leq i \leq \mathsf{len}(s) \implies \\ (\mathsf{substr}(s, 0, i) \notin R \vee \mathsf{substr}(s, i, \mathsf{len}(s) - i) \notin \mathsf{star}(R))\end{array}}$$

Figure 4.18. Negative membership reduction rules, where s, t are terms of String, and R_i's are regular expressions.

The correctness of the rules **R-N-Empty**, **R-N-Sigma** and **R-Concat** follows

from the definition of regular expressions. rempty is a regular language that recognizes the empty language. If a string is not in the set of all characters, its length must be different from 1. Similarly, if a string s is not in a singleton set of t, then s is not equal to t.

The correctness of the rules **R-N-Union** and **R-N-Inter** follows from the definition of regular expressions and logical expressions. If a string is not in the union of languages, it is not in the any of these languages. If a string is not in the intersection of languages, it is at least not in one of them.

If a string is not in the concatenation of two languages, then for every split of the string, either the prefix is not in the first language, or the suffix is not in the second language. Since strings are encoded as a finite sequence of characters, we can split a string by enumerating all possible positions in the string and use substring operations to represent both prefixes and suffixes (see Section 3.3). This is reflected by the rule **R-N-Concat**.

Similarly, if a string is not in the Kleene star of a language (e.g., $\mathcal{L}(\mathsf{star}(R))$), then the string is not the empty string and it is not in $\mathcal{L}(\mathsf{rcon}(R, \mathsf{star}(R)))$. We can use the same argument for the correctness of the rule **R-N-Concat**. Thus, the rule **R-N-Star** is correct.

4.3 Summary

In this chapter, we present our work for handling membership constraints. This work can be considered as an extension of the work in Chapter 3.

Solving arbitrary word equations with length constraints may lead to introduc-

tion of membership constraints, in Chapter 3 we described a relatively naïve method for handling these constraints (by unrolling the Kleene stars via the rule **R-Star**). This is also the main reason why our calculus is refutation incomplete in general. In this chapter, we replaced this rule by a set of more complicated rules. With the help of these new rules, we know that our calculus is a decision procedure for regular membership and length constraints.

The set of rules in this chapter can be divided into three groups: (*i*) rules for consuming a character in a regular expression via the derivative function, (*ii*) rules for splitting regular expressions via the beta function, and (*iii*) rules for computing all possible lengths that are recognized by a regular expression. We also proved that our calculus is a decision procedure when the input only consists of concrete regular membership constraints and length constraints.

In addition, we discussed two extensions of our calculus for: (*i*) solving symbolic membership constraints (note that our rules for word equations will generate this sort of constraints), and (*ii*) handling negative membership constraints lazily.

CHAPTER 5

RELATED WORK

A popular approach for solving string constraints, especially if they involve regular expressions, is to encode them into automata problems. For example, [44] presents an automata-based solver, DPRLE, for matching problems of the form $e \subseteq R$ where, in essence, R is a regular expression over a given alphabet and e is a concatenation of alphabet symbols and string variables. This solver has been used to check programs against SQL injection vulnerabilities. This approach has been improved in later work by generating automata lazily from the input problem without requiring a *priori* length bounds [45]. A comprehensive set of algorithms and data structures for performing fast automata operations to support constraint solving over strings is described in [43]. Generally speaking, there are two sorts of automaton-based approaches:

- *singleton-based* (e.g., [35, 95]) where each transition in the automaton represents a single character, and
- *set-based* (e.g., [91, 92, 45]) where each transition represents a set of characters.

Most of the tools based on these approaches offer very limited support to reason about constraints mixing strings and other data types. Also, automata refinement may act as a kind of bottleneck, even though it is very useful in solving membership

constraints. Further discussion can be found in [38, 55].

A different class of solvers is based on reducing string constraints to constraints in other theories, such as bit-vectors. A successful representative of this approach is the HAMPI solver [51], which is used in a variety of static analysis systems. HAMPI works exclusively with string constraints over fixed-size string variables, and membership constraints from fixed-size context-free languages but with one string variable. Input problems are reduced first to bit-vector problems and then to SAT problems. HAMPI is reimplemented and extended to SHAMPI for translating high-level symbolic computations into SMT queries [90].

An alternative approach (developed to support PEX [87] which is a white-box test generation tool for .NET) targets path feasibility problems for programs using the .NET string library [17]. In this approach, string constraints over a large set of string operators, but with no language membership predicates, are abstracted to linear integer arithmetic constraints and then sent to an SMT solver. Each satisfying solution, if any, induces a fixed-length version of the original string problem which is then solved using finite domain constraint satisfaction techniques.

Yet another related solver, KALUZA [82], lifts some restrictions of HAMPI, and supports unbounded strings and restricted regular expressions. The main algorithm, similar to [17], first uses YICES [30], another SMT solver, for finding a model for the lengths of all string variables, then using the model, sends the string constraints with the additional fixed length constraints to HAMPI. Note that it is possible for YICES to return a model which may fail in HAMPI (see [55] for more detailed discussion). This

communication happens recursively, i.e., when HAMPI returns unsat, KALUZA needs to ask YICES for another model. KALUZA answers unsat only when YICES returns unsat. This approach has some similarities with our finite model finding approach. Thus, it may not terminate in general [82].

The Java String Analyzer (JSA) [25] works on the language of Java string constraints. It first translates Java string constraints into a flow graph, then analyzes this flow graph by converting it into a context-free language. This language is approximated to a regular one with the Mohri-Nederhof algorithm which is then encoded as a multi-level automaton (MLFA). Compared to our work, JSA focuses exclusively on Java string analysis, approximation and automaton conversion, while our approach does not depend on any particular language, and solves string constraints primitively without approximation.

Another idea utilized in [23] is to reduce the theory of strings to the theory of bounded arrays, so that string manipulating functions can be supported. Namely, a constraint logic programming (CLP)-based approach instantiates models over bounded-array-encoded strings. This approach is integrated into the solver EMFTOCSP for solving string constraints. PASS [55] combines ideas from automata and SMT. Similarly to JSA, it handles almost all of Java string operations, regular expressions, and string-number conversions. However, it represents strings as parameterized arrays with symbolic length. This requires solving quantified array constraints. Although the authors provide an algorithm for the quantifier elimination over arrays , the satisfiability of parameterized arrays is undecidable unless it falls

in some particular fragment [18]. Moreover, neither binary code nor source code (of JSA or PASS) is publicly available.

MONA [41] is a string solver supporting monadic second-order logic. Although MONA is an automaton-based approach, it uses Multi Terminal BDDs to represent an automaton. This kind of implementation requires sophisticated engineering techniques (see [52]) which may prevent adding additional theory engines into MONA, and thus prevent solving hybrid constraints. Similarly, PISA [86] is another string solver supporting monadic second-order logic. However, the language of PISA is also restricted, e.g., no binary operations between two variables.

REX [92, 91] is another automaton-based string solver. Unlike the lazy technique [45] where the transition represents an integer interval, Rex encodes strings as symbolic finite automata (SFA) first. The transition of SFA uses a logic predicate to represent a set of candidates. Therefore, the SFA can be further encoded as SMT constraints. In the next round, those SMT constraints are asserted to an SMT solver for a model. This approach provides an efficient encoding for solving membership constraints, although the REX tool currently is not supporting hybrid constraints from other theories.

The work most closely related to our word equation part is the study on Z3-STR [98], a recent string solver developed as an extension of the Z3 SMT solver [28] through its user plug-in interface (the work is further extended by accepting a richer language in S3 [89]). Z3-STR considers unbounded strings with concatenation, substring, replace and length functions and accepts equational constraints over strings as

well as linear integer arithmetic constraints. Its main idea is to have Z3 treat string function and predicate symbols as uninterpreted but monitor the inferences of Z3's equality solver and generate and pass to Z3 selected string theory lemmas as needed. These lemmas, roughly speaking, are used to force the identification of equivalent string terms (e.g., the lemma $s \cdot \epsilon \approx s$ where $\cdot$ is the concatenation operator and ϵ is the empty string), or the dis-identification of terms that Z3 has wrongly guessed to be equal (e.g., $|t| > 0 \implies s \not\approx s \cdot t$). The approach is refutation incomplete because it does not always generate enough axioms to recognize an unsatisfiable problem. At a very high level, our approach is similar, and similarly incomplete, except that it uses a different and more comprehensive set of rules to generate suitable axioms, and so is able to recognize more unsatisfiable cases. Another big difference is that we have devised our solver with the goal of implementing it in an internal, fully integrated theory solver for CVC4, as opposed to an external plug-in, which allows us to leverage several features of the DPLL(T) architecture.

The success of most approaches relies on whether there is an *upper bound* on the length of each string variable, e.g., [74, 51]. We refer them as the theory of bounded strings. However, there is a real demand for a string solver for the theory of unbounded strings. For example, as addressed in [5], a bounded string solver can only be used for the validation but not for the verification of a general policy. There are several proposed approaches for the theory of unbounded strings, e.g., [32, 93]. However, they cannot handle the length constraints, not to mention mixing with other theories. This limits the use of these approaches.

In [50], the authors proposed two criteria for string solvers: modeling cost and accuracy. In addition, they compared four string solvers with their own extensions for modeling some string operations (such as global replacement) in JAVA. Their experiments showed that there is no best solver with respect to their criteria, but the choice of string solvers highly depends on the properties of benchmarks. With respect to CVC4, our modeling cost for a string function is small and our accuracy is high.

CHAPTER 6

CONCLUSION

We have devised an algebra-based calculus for reasoning over a theory of unbounded strings and language memberships with length constraints. Our approach considers strings and regular expressions as primitive data structures and solves constraints without reduction to other theories (e.g., bit-vectors and automata). It is efficient for solving many classes of problems that are dedicated for the applications of formal verification and security vulnerability detection.

Moreover, our proof procedure models the interactions between theory engines, so our string reasoning procedure can be integrated into general multi-theory SMT solvers based on the DPLL(T) architecture. Indeed, we have implemented our algorithm as a built-in theory engine in CVC4. Our initial experimental results indicate that for problems over word equations and length constraints, our approach outperforms the other existing specialized string solvers (with a comparable input language) in terms of correctness, precision and run time. Hence, it also makes CVC4 highly competitive as an underlying constraint solver for verification and security vulnerability detection tools.

Furthermore, we extended the scope of our string solver to support a richer language of string constraints (e.g., contains, replace, index_of) that occur often in real life applications, especially in security vulnerability detection. In the preliminary

implementation work in CVC4, we reduced these commonly used string manipulating functions to our core language in an efficient manner.

At the theoretical level, we have proved the refutation soundness and the solution soundness for our calculus. By utilizing a fair strategy (e.g., the finite model finding), we have proved the solution completeness, that is, our fair procedure is guaranteed to eventually produce a solution for every satisfiable input. In addition, we have proved the termination and the refutation completeness for our calculus over a theory of regular membership with length constraints. Thus, to the best of our knowledge, our procedure is the first decision procedure for this theory.

In our ongoing work, from a theoretical point of view, we plan to identify more fragments where our approach is refutation complete. Note that although the decidability of the full theory is classified as an open problem, it is highly possible that it is undecidable. At the practical level, we are working on the performance optimization based on real applications, which requires us to collaborate with verification and security experts more closely. Currently, our tool is used in several security related projects. With the help of the benchmarks provided by these tools, we may classify the problems, and thus identify the bottleneck in our approach. In addition, we plan to generalize our calculus to the theory of sequences, where a character is a value of some generic sort.

We believe that the approach described in this thesis provides a new idea for string-based formal methods. Notice that most traditional formal method techniques handle string constraints either by ignoring their existence or by abstracting

to some numeric theories. However, given the Von Neumann architecture (which describes how most computers are designed), we believe verification is more appropriately based on reasoning over strings directly, especially when verifying a policy between databases or web-based applications where strings are the only carrier of information.

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukas Holik, Ahmed Rezine, Philipp Rummer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[2] Habib Abdulrab and Jean-Pierre Pecuchet. Solving word equations. *Journal of Symbolic Computation*, 8(5):499–521, 1989.

[3] Habib Abdulrab and Jean-Pierre Pecuchet, editors. *Word Equations and Related Topics*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[4] Alfred Aho and John Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[5] Muath Alkhalaf, Tevfik Bultan, and Jose L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 947–957, Piscataway, NJ, USA, 2012. IEEE Press.

[6] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, March 1996.

[7] Franz Baader and Cesare Tinelli. A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (Townsville, Australia)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 19–33. Springer-Verlag, 1997.

[8] Bahareh Badban and Mohammad Torabi Dashti. Semi-linear parikh images of regular expressions via reduction. In *Proceedings of the 35th International Conference on Mathematical Foundations of Computer Science*, MFCS'10, pages 653–664, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of LPAR'06*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[12] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[13] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[14] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014. (to appear).

[15] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, December 1986.

[16] Nikolaj Bjørner, Vijay Ganesh, R. Michel, and Margus Veanes. An SMT-LIB Format for Sequences and Regular Expressions. In *In SMT workshop 2012*, 2012.

[17] 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: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, pages 307–321. Springer-Verlag, 2009.

[18] Aaron Bradley, Zohar Manna, and Henny Sipma. What's decidable about arrays? In E.Allen Emerson and KedarS. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer Berlin Heidelberg, 2006.

[19] David Brumley, Juan Caballero, Zhenkai Liang, and James Newsome. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.

[20] David Brumley, Hao Wang, Somesh Jha, and Dawn Xiaodong Song. Creating vulnerability signatures using weakest preconditions. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 311–325, 2007.

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

[22] J. Richard Büchi and Steven Senger. Coding in the existential theory of concatenation. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Buchi*, pages 665–670. Springer New York, 1990.

[23] Fabian Büttner and Jordi Cabot. Lightweight string reasoning in model finding. *Software and Systems Modeling*, pages 1–15, 2013.

[24] Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. In *Proceedings of the 5th International Conference on Language and Automata Theory and Applications*, LATA'11, pages 179–191, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[26] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Roy Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[28] 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/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Rolf Drechsler and Bernd Becker. *Binary Decision Diagrams: Theory and Implementation*. Springer, 1998.

[30] Bruno Dutertre and Leonardo De Moura. The YICES SMT solver. Technical report, SRI International, 2006.

[31] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[32] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[33] Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2014.

[34] N.J. Fine and H.S. Wilf. Uniqueness theorem for periodic functions. In *Proceedings of the American Mathematical Society*, volume 16, pages 109–114, 1965.

[35] Xiang Fu and Chung chih Li. A string constraint solver for detecting web application vulnerability. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, SEKE'2010. Knowledge Systems Institute Graduate School, 2010.

[36] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: What's decidable? In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, HVC'12, pages 209–226, Berlin, Heidelberg, 2013. Springer-Verlag.

[37] Wouter Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science*, 411:2987 – 2998, 2010.

[38] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.

[39] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[40] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, pages 109–117. IEEE, 2008.

[41] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '95, pages 89–110, London, UK, UK, 1995. Springer-Verlag.

[42] Pieter Hooimeijer. *Decision procedures for string constraints*. PhD thesis, University of Virginia, 2012.

[43] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, pages 248–262. Springer-Verlag, 2011.

[44] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198. ACM, 2009.

[45] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 377–386. ACM, 2010.

[46] Pieter Hooimeijer and Westley Weimer. StrSolve: solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, 2012.

[47] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[48] Ranjit Jhala and Kenneth L McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 459–473. Springer, 2006.

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

[50] 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.

[51] 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*, pages 105–116. ACM, 2009.

[52] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. In *Revised Papers from the 5th International Conference on Implementation and Application of Automata*, CIAA '00, pages 182–194, London, UK, UK, 2001. Springer-Verlag.

[53] Dexter Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE Computer Society, 1977.

[54] K. Rustan M. Leino. Developing verified programs with dafny. *Ada Lett.*, 32(3):9–10, December 2012.

[55] Guodong Li and Indradeep Ghosh. Pass: String solving with parameterized array and interval automaton. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing*, volume 8244 of *Lecture Notes in Computer Science*, pages 15–31. Springer International Publishing, 2013.

[56] 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 Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[57] M. Lothaire. *Combinatorics on Words*. Cambridge University Press, Cambridge, United Kingdom, 2002.

[58] M. Lothaire. *Applied Combinatorics on Words (Encyclopedia of Mathematics and Its Applications)*. Cambridge University Press, Cambridge, United Kingdom, 2005.

[59] Kenny Zhuo Ming Lu. *XHaskell - Adding Regular Expression Type to Haskell*. PhD thesis, National University of Singapore, 2009.

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

[61] Florin Manea, Robert Mercaş, and Dirk Nowotka. Fine and wilf's theorem and pseudo-repetitions. In *Proceedings of the 37th International Conference on Mathematical Foundations of Computer Science*, MFCS'12, pages 668–680, Berlin, Heidelberg, 2012. Springer-Verlag.

[62] Yuri Matiyasevich. Hilbert's tenth problem and paradigms of computation. In *Proceedings of the First International Conference on Computability in Europe: New Computational Paradigms*, CiE'05, pages 310–321. Springer-Verlag, Berlin, Heidelberg, 2005.

[63] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, October 1979.

[64] J. Nielsen. Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. *Mathematische Annalen*, 78(1):385–397, 1917.

[65] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[66] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[67] OWASP. OWASP Top 10 2013 project. Technical report, The Open Web Application Security Project (OWASP), June 2013. (Available at `http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf`).

[68] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.

[69] Sam Owre and Natarajan Shankar. A brief overview of pvs. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 22–27, Berlin, Heidelberg, 2008. Springer-Verlag.

[70] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, October 1966.

[71] Wojciech Plandowski. Satisfiability of word equations with constants is in nexptime. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 721–725, New York, NY, USA, 1999. ACM.

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

[73] Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 467–476, New York, NY, USA, 2006. ACM.

[74] 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, New York, NY, USA, 2012. ACM.

[75] James Renshaw. Monoids, acts and categories: With applications to wreath products and graphs: A handbook for students and researchers. *Semigroup Forum*, 66(3):489–490, 2003.

[76] Andrew Reynolds. *Finite Model Finding in Satisfiability Modulo Theories*. PhD thesis, The University of Iowa, December 2013.

[77] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification (St Petersburg, Russia)*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer, 2013.

[78] Grigore Rosu and Mahesh Viswanathan. Testing extended regular language membership incrementally by rewriting. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 499–514. Springer Berlin Heidelberg, 2003.

[79] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[80] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 2: Linear Modeling: Background and Application*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[81] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 3: Beyond Words*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[82] 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*, pages 513–528. IEEE Computer Society, 2010.

[83] K.U. Schulz, editor. *Word Equations and Related Topics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[84] Martin Sulzmann. Playing with regular expressions: Intersection. Technical report, IT University of Copenhagen, 2008.

[85] Martin Sulzmann and Kenny Zhuo Ming Lu. POSIX regular expression parsing with derivatives. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 203–220, 2014.

[86] 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.

[87] Nikolai Tillmann and Jonathan Halleux. Pex - white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

[88] Cesare Tinelli and Calogero Zarba. Combining non-stably infinite theories. In I. Dahn and L. Vigneron, editors, *Proceedings of the 4th International Workshop on First Order Theorem Proving, FTP'03 (Valencia, Spain)*, volume 86.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[89] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In Moti Yung and Ninghui Li, editors, *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[90] Richard Uhler and Nirav Dave. Smten: Automatic translation of high-level symbolic computations into smt queries. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 678–683. Springer Berlin Heidelberg, 2013.

[91] Margus Veanes. Applications of symbolic finite automata. In *Proceedings of the 18th International Conference on Implementation and Application of Automata*, CIAA'13, pages 16–23, Berlin, Heidelberg, 2013. Springer-Verlag.

[92] Margus Veanes, Nikolaj Bjørner, and Leonardo De Moura. Symbolic automata constraint solving. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 640–654, Berlin, Heidelberg, 2010. Springer-Verlag.

[93] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[94] Clay Wilson. Computer attack and cyberterrorism: Vulnerabilities and policy issues for congress. Technical report, Congressional Research Service, The Library of Congress, Congressional Research Service, The Library of Congress, April 2005. Order Code RL32114.

[95] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 154–157. Springer Berlin Heidelberg, 2010.

[96] Fang Yu, Tevfik Bultan, and Ben Hardekopf. String abstractions for string verification. In Alex Groce and Madanlal Musuvathi, editors, *Model Checking Software*, volume 6823 of *Lecture Notes in Computer Science*, pages 20–37. Springer Berlin Heidelberg, 2011.

[97] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit propagation. In *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH96), Fort Lauderdale (Florida USA)*, pages 166–169, 1996.

[98] 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, pages 114–124, New York, NY, USA, 2013. ACM.

[99] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, Sep 1978.