REASONING OVER STRINGS AND OTHER UNBOUNDED DATA STRUCTURES

TRINH MINH THAI

(BCompSc. Hons.)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Supervisor: Professor Joxan Jaffar

Examiners: Professor Dong Jin Song

Assistant Professor Prateek Saxena

Dr Nikolaj S. Bjørner, Microsoft Research, Redmond

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2017

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

TRINH MINH THAI

28 April 2017

Acknowledgements

My deepest gratitude is to my parents for raising a stubborn child like me. They play a major role in turning me into a man who is now open to others' opinions while still having a strong belief in himself. I am always proud of having unconditional love and support from my parents and my family.

Like many of us, my early days of doing research is not fun. I did cope with many difficulties in the way of searching for my interests. Moreover, I also struggled with starting a new and independent life overseas. Whenever thinking about those days, I feel indebted to Professors Chin Wei Ngan and Truong Anh Hoang, who not only gave me the opportunity to start my wonderful adventure in Singapore but also took care of me both in work and life.

The fun part came late in the second year of my PhD study. That was the first time I had found one of my research interests, when attending a program security class by Prateek Saxena. That was also the time I started doing research on my own. The greatest thing is that meanwhile I still have received full support from my supervisor Professor Joxan Jaffar and my colleague Chu Duc Hiep. I am grateful to them for teaching me a lot of things. Without them, I might not achieve what I have had now.

Finally, I would like to thank my thesis examiners Nikolaj Bjørner, Professor Dong Jin Song, and Prateek Saxena, and the anonymous reviewers of my three publications for their useful comments. I also thank Ta Quang Trung for pointing out a confusing point in the PLDI'15 paper so that I can clarify it in the thesis. I also would like to thank all of my friends and colleagues for sharing with me unforgetable moments in my PhD life.

Contents

SUMMARY

Data structures play a central role in software development. While developing proper data structures is not easy, reasoning about them is even harder. The difficulty comes from their typical characteristic: the *unboundedness* of the data structures and/or the loops manipulating them. This makes two following fundamental issues more severe: *inter-dependence* between structures and data values, and complicated *interactions* between different data structures.

In this thesis, we consider the problem of reasoning about unbounded data structures such as strings, linked lists, trees. Specifically, we propose systematic techniques for the satisfiability problem of string constraints, and the entailment proving problem for heap-allocated data structures. These two problems are of great interest: for example, while the former is important for security analysis of web applications, the latter is important for automated verification of imperative programs.

The first technique is to implement *lazy reasoning* methodology. Its introduction is to mitigate the problem of combinatorial explosion in searching for a solution of the input constraints. Specifically, we incrementally reduce recursive predicates, which are used to represent string operations, via splitting (and/or unfolding) process, until their subparts are bounded with constant strings/characters to be consumed. We have applied this technique in building an *efficient* string solver. While modern string solvers exist, they suffer in one way or another: (1) the constraint language may not be expressive enough (even though the solver is fast); or (2) the solver may not be fast enough to accommodate realistically large programs. Thanks to lazy reasoning, we now have a fast symbolic string solver to support an expressive language, thus opening doors for future development of comprehensive frameworks for vulnerability detection in web applications.

Since lazy reasoning does not address non-termination in solving string constraints, we next propose a novel method, namely, *progressive reasoning*. The key feature of the new algorithm is a pruning method on the subproblems, in a way that is *directed*. More specifically, our algorithm detects non-progressive scenarios with respect to a criterion of minimizing the "lexicographical length" of the returned solution, if a solution in fact exists. Informally, in the search process based on reduction rules, we can soundly prune a subproblem when the answer we seek can be found more efficiently elsewhere. If a subproblem is deemed non-progressive, it means if the original input formula is satisfiable, then another

satisfiable solution of shorter "length" will be found. If, on the other hand, the input formula is unsatisfiable, then any pruning is obviously sound. A technical challenge we will overcome is that at the point of pruning, the satisfiability of the input formula is unknown. Experimental evaluations show the promising results of our new string solver in dealing with non-termination in string solving.

Finally, we propose a general method that includes *inductive reasoning* for entailment proving. It aims to address non-termination in proving heap-allocated data structure properties. The challenge is how to use induction correctly and avoid erroneous proof arising from a form of *circular reasoning*. Our method is able to use *dynamically generated* formulas as induction hypotheses, and to enforce an *anti-circular* condition so that any application of an induction step is guaranteed to be correct. The state-of-the-art methods are often unable to prove relationship between different data structures (e.g. to prove that a sorted list is a list). As a result, they would not be able to automatically verify a large class of programs. Inductive reasoning helps us to close such remaining gap in existing systems. More importantly, it also gets us back the power of compositional reasoning in dealing with user-defined recursive predicates that are used to represent data structures properties.

List of Tables

List of Figures

Chapter 1

Introduction

Data structure is an organization of information, usually in memory, for better algorithm efficiency [Pieterse and Black, 2004]. They are usually associated with operations that can be performed on them and the computational complexity of each operation. Different kinds of data structures are suitable for different kinds of operations/functions. For example, relational databases commonly use B-tree indexes for data retrieval [Powell, 2006], while compiler implementations use hash tables for looking up identifiers [Aho *et al.*, 1986].

Developing proper data structures is not easy; reasoning about them is even harder, especially when the size of the data structures and/or the loops manipulating them is unbounded. We refer to them as *unbounded data structures*. For example, linked list is an unbounded data structure since a linked list contains an unspecified number of nodes [Cormen *et al.*, 2001]. Another example is the string data type in JavaScript language: the `replace` operation can replace an unspecified number of matches of a pattern in a string by a replacement [Mozilla, 2016]. In both cases, it is necessary to assume the sizes of the data structures (e.g., the linked list and the string) to be unknown in order to precisely capture their properties.

1.1 Reasoning over Unbounded Data Structures

In modern software, unbounded data structures are not only ubiquitous, they also play a critical part: their improper use may affect the software security and safety. However, reasoning about them is not trivial. In this section, we first present motivations for reason-

ing over strings and other unbounded data structures, then we discuss challenges for such reasoning.

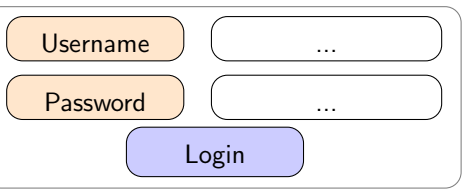
1.1.1 Motivating Examples

To show the importance of reasoning about unbounded data structures, let us use two illustrative examples: web applications and heap-manipulating programs.

1.1.1.1 Web Applications

According to the Open Web Application Security Project [OWASP, 2013], the most serious web application vulnerabilities include: (#1) Injection flaws (such as SQL injection) and (#3) Cross Site Scripting (XSS) flaws. Both vulnerabilities involve string-manipulating operations and occur due to inadequate sanitisation and inappropriate use of input *strings* provided by users.

To illustrate more clearly the importance of reasoning about strings in ensuring the security of web applications, we use a simple example of a log-in form in Figure 1.1. The user needs to fill in the client-side form, by providing a username to the HTML input element `username` and a password to the HTML input element `password`. When the `Login` button is clicked, the browser will submit the string values of these two fields to the server.




```
...
<html>
<form name="loginForm" action="/Login">
    Username: <input type="text" name="username" size="64" />
    Password: <input type="text" name="password" size="64" />
    <input type="submit" value="Login" />
</form>
...
</html>
```

Figure 1.1: A login form along with its simplified HTML code

Let us call the vulnerable website `www.vulnerable.site` and look at two following security questions.

Security Question 1: One interesting security question is whether this web page is vulnerable to an SQL injection. More specifically, can the following PHP code, with an appropriate instantiation for string variable `$usr`, be executed on the server side, leading to an attack?

```
$usr = $_POST['username'];
$pwd = $_POST['password'];
$stm = "SELECT tbl1 where username='$usr' and password='$pwd'";
$result = mysql_query($stm);
```

The answer is *yes*. For example, attackers can create an SQL injection by using the username `' OR 1=1--` (and an arbitrary password) in order to construct an SQL query `$stm`

```
SELECT tbl1 where username='' OR 1=1--' and password='...'
```

Because of the comment notation (`--`), the actual SQL query is

```
SELECT tbl1 where username='' OR 1=1
```

which will select all the rows of the table `tbl1`. This eventually allows the attackers to steal the information of all the usernames and passwords from the table `tbl1`.

Security Question 2: Now suppose there is also a script `welcome.cgi`, which takes one parameter `name`, in the current site `www.vulnerable.site`. This script reads part of the HTTP request:

```
GET /welcome.cgi?name=Minh%20Thai HTTP/1.0
Host: www.vulnerable.site
```

and echoes the current username back to the response page:

```
<HTML>
<Title>Welcome!</Title>
Hi Minh Thai <BR>
Welcome to our system
...
</HTML>
```

Similarly to the previous example, we can ask if the site is vulnerable to XSS attacks. The answer is *yes*. In fact, the attacker is able to prepare a link such as

```
http://www.vulnerable.site/welcome.cgi?name=<script>window.open
("http://www.attacker.site/collect.cgi?cookie="%2Bdocument.cookie)</script>
```

and lures the victim client into clicking that link (e.g. by phishing them). This link causes the Web browser of the victim to access the site `www.vulnerable.site` and to invoke the vulnerable script `welcome.cgi`. The victim, upon clicking the link, will generate a request to `www.vulnerable.site`, as follows:

```
GET /welcome.cgi?name=<script>window.open("http://www.attacker.site/
collect.cgi?cookie="%2Bdocument.cookie)</script> HTTP/1.0
Host: www.vulnerable.site
```

and the site's response would be:

```
<HTML>
<Title>Welcome!</Title>
Hi <script>window.open("http://www.attacker.site/collect.cgi?cookie=
"+document.cookie)</script> <BR>
Welcome to our system
...
</HTML>
```

The victim's Web browser, immediately upon loading this page, would execute the embedded JavaScript and send a request to the `collect.cgi` script in `www.attacker.site`, with the value of the cookies of `www.vulnerable.site` that the browser already has. This compromises the cookies of `www.vulnerable.site` that the victim client has and allows the attacker to impersonate the victim.

In summary, it can be seen that strings are ubiquitous in web applications. A web application usually takes string values as input, manipulates them, and then uses them to construct database queries. As such, if the string inputs are not appropriately used or not fully sanitized, web applications will be vulnerable to serious attacks. Therefore, it is necessary to reason about strings in order to detect security vulnerabilities and ensure the security of web applications.

1.1.1.2 Heap-manipulating Programs

In imperative programs, data structures such as linked lists are used frequently. The purpose of using these kinds of data structures is to carry out memory (de)allocation while the program is still running. Because the allocated memory is in the heap, we call a program that use these kinds of data structures a heap-manipulating program.

The improper use of heap-allocated data structures can be a source of bugs related to memory (de)allocation [Wikipedia, 2016]. These can include security bugs or program crashes, often due to segmentation faults. The most common errors are as follows:

- Memory leaks: Failure to deallocate memory using `free` command leads to buildup of non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.
- Logical errors: All allocations must follow the same pattern: allocation using `malloc`, usage to store data, deallocation using `free`. Failures to adhere to this pattern, such as memory usage after a call to `free` (a.k.a. use-after-free) or before a call to `malloc` (a.k.a. wild pointer), calling `free` twice (a.k.a. double free), etc., usually cause a segmentation fault and result in a program crash.

These errors can be transient and hard to debug. For example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

Importantly, these improper uses of heap-allocated data structures can lead to critical security vulnerabilities. For example, calling `free` twice on the same memory address potentially modifies unexpected memory locations. When a program calls `free` twice with the same argument, the program's memory management data structures become corrupted. This corruption can cause the program to crash or, in some circumstances, cause two later calls to `malloc` to return the same pointer. If `malloc` returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack [Microsoft, 2013].

As another example, dangling/wild pointer bugs frequently become security holes. For example, if the pointer is used to make a virtual function call, a different address (possibly pointing at exploit code) may be called due to the `vtable` pointer being overwritten.

Alternatively, if the pointer is used for writing to memory, some other data structure may be corrupted. Even if the memory is only read once the pointer becomes dangling, it can lead to information leaks (if interesting data is put in the next structure allocated there) or to privilege escalation (if the now-invalid memory is used in security checks). When a dangling pointer is used after it has been freed without allocating a new chunk of memory to it, this becomes known as a "use after free" vulnerability [Dalci, 2012]. For example, CVE-2014-1776 is a use after free vulnerability being used by zero-day attacks by an advanced persistent threat [Xiaobo *et al.*, 2014].

1.1.2 Main Challenges

Though there are clear motivations for reasoning over unbounded data structures, such reasoning is difficult. The difficulty comes from the unboundedness of the data structures and/or the loops manipulating them. The unboundedness requires us to assume the size of the data structures to be unknown in order to precisely capture their properties. Therefore, we are not able to do concrete reasoning about unbounded data structures. This makes two following fundamental issues more severe.

The first fundamental issue is inter-dependence between structures and data values. For instance, to define a binary search tree (BST), we have to compare the data value of a parent node with the values of its child nodes in the left and right subtrees. As a result, to analyze the shape (or structure) of a BST, it is inevitable to handle the relationship between its data values. In other words, it will be imprecise to reason about the structures and the data values separately from each other.

The second fundamental issue is complicated interactions between different data structures. These interactions happen when we split a string into sub-strings, concatenate a list of sub-strings into one, or copy data values from a tree to a new one, etc. This gives rise to the need for reasoning about the relationship between different unbounded data structures.

1.2 The Satisfiability-Based Approach

The last decades have witnessed a lot of research work on program reasoning over unbounded data structures [Wies *et al.*, 2007; Bouajjani *et al.*, 2009b; Srivastava, 2010; Itzhaky *et al.*, 2013]. One of successful approaches is the satisfiability-based where the center of program

reasoning tools is SAT/SMT solvers [De Moura and Bjørner, 2011; Barrett *et al.*, 2009]. An advantage of this paradigm is the re-usability. Specifically, one can develop an off-the-shelf SMT solver which can be used by different program reasoning tools.

To illustrate, in program reasoning for web applications, the state-of-the-art technique is dynamic symbolic execution (DSE). Its main purpose is to avoid false positives, but still preserve high code coverage. Some examples of recent works based on DSE are [Godefroid *et al.*, 2008; Halfond *et al.*, 2009; Kiezun *et al.*, 2009b; Chaudhuri and Foster, 2010; Saxena *et al.*, 2010; Bisht *et al.*, 2010; Bisht *et al.*, 2011; Ghosh *et al.*, 2013; Sen *et al.*, 2013; Maras *et al.*, 2013; Wang *et al.*, 2013; Jensen *et al.*, 2013; Bucur *et al.*, 2014]. These works employ both concrete and symbolic execution to automatically and systematically generate tests in order to expose vulnerabilities in web applications. DSE for automated test generation involves instrumenting and *concolically* running a program while collecting path constraints on the inputs. Then it attempts to derive new inputs using an SMT solver with the hope to steer next executions toward new program paths. For vulnerability detection, DSE combines the derived path constraints with the specifications for attacks, often given by the security experts, to create queries for the SMT solver. In short, the problem of vulnerability detection will be reduced into the problem of deciding if a constraint formula is satisfiable. Furthermore, if the formula is satisfiable, we should be able to generate a solution/model (in order to derive new inputs for exploring new paths or to generate attack inputs that exploit the vulnerabilities).

Another example is automated verification techniques for imperative programs [Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Piskac *et al.*, 2013; Pek *et al.*, 2014]. Typically, pre/post conditions are specified for each method/function (and an invariant given for each loop) before the reasoning system automatically checks if each given program code is correct w.r.t. the given annotations. This problem basically will result in proving entailments[1] of the form $A \models B$, where A, B involve recursive predicates that represent data structures such as linked lists. In fact, we can rephrase it as a problem of proving that $A \wedge \neg B$ is unsatisfiable, which is merely a satisfiability problem. The difference here is that we do not need to generate a solution/counter-example in the case $A \wedge \neg B$ is satisfiable.

[1] We will explain this reduction step in Chapter 2.

1.3 Thesis Contributions

In this thesis, we address the problem of deciding the satisfiability of constraints over recursive predicates that are used to represent unbounded data structures. We first propose three systematic techniques to reason about recursive predicates. They are lazy reasoning, progressive reasoning for strings (e.g. in web applications), and inductive reasoning for heap-allocated data structures (e.g. in imperative programs). Finally, we demonstrate their applications in the satisfiability-based program reasoning tools in order to ensure the program safety and security.

1.3.1 Lazy Reasoning

When searching for a solution of the input formula containing constraints over recursive predicates, one way to discharge recursive predicates is to generate all of their reduction possibilities. This is also called generate-and-test technique (e.g. [Saxena *et al.*, 2010]). For example, suppose we have the string formula

$$p_1 \in /(ab)^\star/ \wedge p_2 \in /(bc)^\star/ \wedge p_1 \cdot p_2 = \text{“}ababababababcc\text{”}$$

which requires p_1 and p_2 belong to the regular expression $/(ab)^\star/$ and $/(bc)^\star/$ respectively, and their concatenation is equal to “*ababababababcc*”. Because the length of the string “*ababababababcc*” is 14, we can infer that the sum of the length of p_1 and the length of p_2 is equal to 14. So a satisfying solution for the length of p_1 and p_2 must be a pair in $\{\langle 0; 14\rangle, \langle 2; 12\rangle, \langle 4; 10\rangle, \langle 6; 8\rangle, \langle 8; 6\rangle, \langle 10; 4\rangle, \langle 12; 2\rangle, \langle 14; 0\rangle\}$. As a result, we need to test 8 cases to find out that the input formula is unsatisfiable. However, let us not have the impression that, in general, the number of satisfying solutions for the string lengths should be of this linear complexity. In fact, practical applications involve many string variables, generate-and-test approach would easily suffer from a combinatorial explosion.

Lazy reasoning is introduced to mitigate that combinatorial explosion problem by reducing recursive predicates *on demand*. For heap-allocated data structures, lazy reasoning is implemented via Unfold-and-Match technique. Basically, the proof search proceeds by repeatedly applying (un)folding strategies, until all the reduced predicates in the RHS of the obligation can be cancelled out by corresponding predicates in the LHS via a simple, non-recursive, matching method [Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Piskac *et al.*, 2013; Pek *et al.*, 2014].

Though Unfold-and-Match technique has been introduced before for the purpose of proving entailments of the form $A \models B$ (i.e., it aims at proving the unsatisfiability of $A \wedge \neg B$), it does not work in tandem with the process of searching for a solution. Therefore, in this thesis, we propose the Unfold-and-Consume technique for string solving. Essentially, we incrementally reduce recursive predicates that are used to represent string operations via splitting (and/or unfolding) process, until their subparts are bounded with constant strings/characters to be consumed.

We have implemented the Unfold-and-Consume technique into the string solver, namely S3. This new solver is composed of a string theory plugged into the state-of-the-art Z3 SMT solver [De Moura and Bjørner, 2008b]. The result is published in the paper [Trinh *et al.*, 2014]. There are three key contributions. First, S3 is *expressive*. Specifically, it is the first to handle unbounded regular expressions in the presence of length constraints, and express precisely high-level string operations, which ultimately enables a more accurate string analysis.

Second, S3 is *robust*. This means that S3 is able to provide definitive answers to a new level, far beyond the state-of-the-art. This in turn means we can detect more vulnerabilities and more bugs. We demonstrate it with two case studies:

- The first is to compare with Kaluza – the core of Kudzu [Saxena *et al.*, 2010] – a JavaScript symbolic execution framework. We show that S3 is several times faster, and helps detect many more paths that reach the critical sink, that is, paths that are vulnerable.

- The second is to compare with Z3-str [Zheng *et al.*, 2013]. We show S3 reasons about length constraints much more effectively than Z3-str. This leads to a large increase in applicability to web programs, because length constraints are widely used.

Third, S3 is *efficient*, and one key reason is that it is *incremental*. Our algorithm for string theory is designed in an incremental fashion driven by the try-and-backtrack procedure of the Z3 core, so that given a set of input constraints, we perform incremental reduction for string variables until the variables are bounded with constant strings/characters. Another technical challenge is how to reason, effectively and efficiently, about the Kleene star and high-level operations such as `replace` (in its most general usage), of which the semantics are by nature recursively defined. The gist of our proposal is the encodings using recursively-

defined functions, on which we can incrementally reason: by lazily *unfolding* them.

Though lazy reasoning has solved the problem of combinatorial explosion in entailment proving and string solving, it does not address non-termination issues. As such, we next propose two novel techniques in order to address them: progressive reasoning for string solving and inductive reasoning for entailment proving.

1.3.2 Progressive Reasoning

Unfold-and-Consume technique has shown very promising results (e.g. as in [Trinh *et al.*, 2014]). However, because its main purpose is vulnerability detection, i.e., generating attack inputs for each satisfiable query, and that every query is invoked with a timeout limit, there was less emphasis on the detection of *unsatisfiable* queries. By contrast, in the setting of program verification, or in using verification technologies to speed up concolic testing [Jaffar *et al.*, 2013; Avgerinos *et al.*, 2014], the problem of determining unsatisfiability becomes paramount. In short, we can no longer depend on a timeout, and must seek a *terminating* algorithm as far as possible.

This motivates our proposal of a progressive search algorithm whose goal is to determine if a string formula is unsatisfiable, and if not, to be able to generate a solution for it. The result is published in the paper [Trinh *et al.*, 2016]. The key feature of our algorithm is a *pruning method* on the subproblems, in a way that is *directed*. More specifically, our algorithm aims to detect non-progressive scenarios with respect to a criterion of minimizing the "lexicographical length" of the returned solution, if a solution in fact exists. Informally, in the search process based on reduction rules, we can soundly prune a subproblem when the answer we seek can be found more efficiently *elsewhere*. If a subproblem is deemed non-progressive, it means if the original input formula is satisfiable, then another satisfiable solution of shorter "length" will be found. If, on the other hand, the input formula is unsatisfiable, then any pruning is obviously sound. A technical challenge we will overcome is that at the point of pruning, the satisfiability of the input formula is *unknown*.

An additional important feature of our algorithm is applicable only when the input formula is unsatisfiable. Here, we want to produce a set of *conflict clauses*, a generalization of the input formula, that is now known to be unsatisfiable. The benefits of such learning is of course well-known. It is, for example, at the heart of the attractiveness of SMT solvers. However, the key technical challenge is, how conflict clause learning can work in tandem

with the pruning of non-progressive formulas, because at the time of pruning, again, the unsatisfiability of the input formula is unknown.

Finally, we present an experimental evaluation with two case studies. First is on the well-known Kudzu benchmark [Saxena *et al.*, 2010] where we show that (a) our new algorithm surpasses four state-of-the-art solvers in its ability to detect unsatisfiable formulas or generate a model in satisfiable formulas (and in good running time), and (b) the number of unsatisfiable cores is very small, thus paving the way to accelerate the consideration of large collections of formulas. The second case study considers web applications used in the Jalangi framework [Sen *et al.*, 2013], and shows how we can deal with the `replace` operation in string formulas. No other system has been demonstrated on this class of problems, and thus the purpose of our evaluation is simply to show that we are applicable.

1.3.3 Inductive Reasoning

Given an input obligation, a proof system will apply reduction rules to transform that obligation into new obligations, which are also in the form of entailment. An entailment proof, using Unfold-and-Match, succeeds when we find a sequence of successive applications of these transformation steps that produce a final formula which is *obviously* provable. This usually means that either (1) there is no recursive predicate in the RHS of the proof obligation and a direct proof can be achieved by consulting some generic SMT solver; or (2) no special consideration is needed on any occurrence of a predicate appearing in the final formula. For example, if $\mathtt{p}(\tilde{u}) \wedge \cdots \models \mathtt{p}(\tilde{v})$ is the formula, then this is obviously provable if $\tilde{u}$ and $\tilde{v}$ were *unifiable* (under an appropriate theory governing the meaning of the expressions $\tilde{u}$ and $\tilde{v}$). In other words, we have performed "formula abstraction" [Madhusudan *et al.*, 2012] by treating the recursively defined term $\mathtt{p}()$ as *uninterpreted*.

A key feature that is missing from the Unfold-and-Match methodology is the ability to prove by *induction*, which is often required in verification of practical examples [Berdine *et al.*, 2005]. Without inductive reasoning, Unfold-and-Match (folding/unfolding together with formula abstraction) *cannot* handle proof obligations involving *unmatchable* predicates. Specifically, in such obligations, there exists a recursively defined predicate in the RHS which cannot be transformed, via folding/unfolding, to one that is unifiable with some predicate in the LHS.

In this thesis, we propose a general proof method for recursive predicates that includes

reasoning by induction. The challenge is how to use induction correctly and avoid erroneous proof arising from a form of *circular reasoning*. Our method is able to use *dynamically generated* formulas as induction hypotheses, and to enforce an *anti-circular* condition so that any application of an induction step is guaranteed to be correct. We shall see that our method is very different from that in traditional entailment proving systems where, after having chosen an induction tactic, the system will then search for appropriate induction variable(s) with a well-founded measure and appropriate induction hypotheses. In our framework, the predicates are defined by general recursive rules, without any explicit restriction to any well-founded orderings, and includes a domain of discourse that captures the mutable heap and properties of separation.

The result is published in the paper [Chu *et al.*, 2015]. Specifically, our contributions are as follows. First, we automatically and efficiently *discharge* all commonly-used lemmas, extracted from a number of benchmarks used by other systems. These systems cannot automatically discharge such lemmas, but simply accept them as true facts.

Second, we demonstrate that with our proof method, the common usage of lemmas can be *avoided*. This is because the properties of interest are covered by our method. In contrast, these properties cannot be discharged by the other systems without using lemmas. The impact of this is twofold. First, it means that for proving practical (but small) programs, the users are now free from the burden of providing custom user-defined lemmas. Second, it significantly boosts up the performance, since lemma applications, coupled with folding/unfolding, often induce a large search space.

Lastly, the proposed proof method gets us back the power of compositional reasoning in dealing with user-defined recursive predicates. While we have not been able to identify *precisely* the class where our proof method would be effective, we do believe that its potential impact is huge. One important subclass that we can handle effectively is when both the antecedent and the consequent refer to the same structural shape but the antecedent simply makes a *stronger* statement about the values in the structure (e.g., to prove that a sorted list is also a list, an AVL tree is also a binary search tree, a list consists of all data values 999 is one that has all positive data, etc.).

1.4 Thesis Organization

In Chapter 2, we provide the preliminary background on satisfiability solving and entailment proving. We also discuss the existing Unfold-and-Match technique, and its application in verifying heap-allocated data structure properties. Chapter 3 presents our lazy reasoning technique for string solving, that is the so-called Unfold-and-Consume. We present our progressive reasoning for string solving in Chapter 4 and inductive reasoning for entailment proving in Chapter 5. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Preliminaries

In this chapter, we present the preliminary background of the satisfiability problem for string theory, which is important for security analysis of web applications, and the entailment proving problem for user-defined recursive predicates, which is important for automated verification of imperative programs. We extend the previous discussion in Chapter 1 on the reduction from web security analysis to string solving and the reduction from automated program verification into entailment proving. For the former, we also present the constraint language to suffice to analyze web applications, and discuss theoretical results on various theories over strings. For the latter, we also introduce the logic for dealing with the heap, and discuss the state-of-the-art proof techniques, namely Unfold-and-Match, for heap-allocated data structures.

2.1 String Solving for Web Security Analysis

Symbolic string solving plays an important role in security analysis of web applications. To explain why, let us look at *dynamic analysis* which involves testing an application as a closed entity with a set of concrete inputs. Its main disadvantage is of course that it is not a complete method. For example, some program paths may only be executed if certain inputs are passed as parameters to the application, but it is very unlikely that a dynamic analyzer can exhaustively test an application with all possible inputs. For web applications, the problem is even more severe since dynamic analysis needs to take into account not only the value space (i.e., how the execution of control flow paths depends on input values),

but also an application's event space (i.e., the possible sequences of user-interface actions). As a result, there is in general an impractical number of execution paths to systematically explore, leading to the "low code coverage" issue of dynamic analysis.

A standard approach to have good or complete coverage is static analysis. However, the problem here is the existence of false positives, arising from an over-approximation of the program's behavior. Recent works to avoid false positives, but still preserve high code coverage, are based on *dynamic symbolic execution* (DSE). Some examples are [Saxena *et al.*, 2010; Bisht *et al.*, 2010; Bisht *et al.*, 2011; Kiezun *et al.*, 2009b; Emmi *et al.*, 2007; Sen *et al.*, 2005; Godefroid *et al.*, 2005; Godefroid *et al.*, 2008; Halfond *et al.*, 2009; Ghosh *et al.*, 2013; Sen *et al.*, 2013; Chaudhuri and Foster, 2010; Bucur *et al.*, 2014; Maras *et al.*, 2013; Wang *et al.*, 2013; Jensen *et al.*, 2013]. These approaches employ both concrete and symbolic execution to automatically and systematically generate tests in order to expose vulnerabilities in web applications. DSE for automated test generation involves instrumenting and *concolically* running a program while collecting path constraints on the inputs. Then it attempts to derive new inputs – using an SMT (Satisfiability Modulo Theories) solver – with the hope to steer next executions toward new program paths. For vulnerability detection, DSE combines the derived path constraints with the specifications for attacks, often given by the security experts, to create queries for the SMT solver.

	Dynamic Analysis	DSE
Code Coverage	Potentially Low	High
False Positives	Low	Low
Executable Paths (EPs)	Unlikely to cover all EPs	Likely to cover all EPs

Table 2.1: Dynamic Analysis versus DSE

In fact, there is a strong connection between an effective vulnerability detection framework and symbolic string solving. As shown in Table 2.1, DSE achieves higher code coverage. However, because not all path executed by DSE are guaranteed to be executable, to avoid false positives we must be able to *decide* if a (symbolic) path constraint is satisfiable or not. Thus a powerful SMT solver, capable of handling symbolic string variables, is the *key* to achieve efficient analyses with high code coverage and low false positives.

To explain more the way DSE detects possible vulnerabilities, in comparison with typical dynamic analyses, let us use a modified version of the example in Figure 1.1. Instead, we now have the following PHP code at the server side:

```
function endsWith($str, $sub) {...}
$usr = $_POST['username'];
$pwd = $_POST['password'];
if (endsWith($usr, "@nus.edu.sg")) {
  $stm = "SELECT tbl1 where username='$usr' and password='$pwd'";
  $result = mysql_query($stm);
}
```

Since a dynamic analysis is essentially black-box testing, it has no knowledge about the code. Thus, it is possible that the dynamic analyzer does not test with usernames that end with `@nus.edu.sg`, and subsequently, cannot detect SQL injection. In contrast, DSE, which can be seen as white-box testing, enables us to attempt all execution paths by generating two path constraints, corresponding to the two program paths of the PHP code fragment.

After symbolically executing the program, DSE frameworks such as [Saxena *et al.*, 2010] will combine its results with the specifications for attacks, given by the security experts, to create queries for the constraint solver. The specifications, often come in form of assertions, are some (regular) grammars encoding a set of strings that would constitute an attack against a particular sink. If the constraint solver finds a solution to a query, then this represents an attack that can reach the critical sink and exploit a code injection vulnerability. For example, with the specification to assert if the username contains `' OR 1=1--`, we can in fact generate the input

```
' OR 1=1--@nus.edu.sg
```

that leads to an SQL injection.

In summary for this subsection, DSE, presently the state-of-the-art in vulnerability detection, is intimately tied to being able to provide definitive answers for the derived constraint queries. In the case of web applications, since the constraints often concern string variables, symbolic string solving is thus the key to detect vulnerabilities in this class of applications.

2.1.1 What Constraint Language Do We Need?

We first argue that a pure string language does not suffice to analyze web applications. This is due to the fact that non-string operations (e.g., boolean, arithmetic constraints) are

also widely used in web applications. Moreover, their use is often intertwined with string operations, such as in the case of string length — a string-to-integer constraint. Reasoning about strings and non-strings *simultaneously* is thus necessary. In other words, we need to deal with a *multi-sorted* theory which includes, at least, strings and integers.

To amplify this point, let us now state some statistics from a comprehensive study of practical JavaScript applications [Saxena *et al.*, 2010]. Constraints arising from the applications have an average (per benchmark query) of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. The largest fraction are for operations like `indexOf`, `length` (78%). A significant fraction of the operations, including `substring` (5%), `replace` (8%), and `split`, `match` (1%). Of the `match`, `split` and `replace` operations, 31% are based on regular expressions. Operations such as `replace` and `split` give rise to new strings from the original ones, thereby giving rise to constraints involving *multiple* string variables.

To summarize, constraints of interest are either non-strings (e.g., bool-sort, int-sort and particularly length constraints) or strings such as: string equations, membership predicates and high-level string operations, which are over multiple string variables.

2.1.2 The Satisfiability Problem for Theories over Strings

To give the readers a sense of the hardness of solving constraints of the above language, let us give a brief review on theoretical works on the satisfiability problem for different theory fragments over strings.

In his original paper, Quine [Quine, 1946] showed that the first-order theory of string equations (i.e., quantified sentences over Boolean combination of word equations) is undecidable. Due to the expressibility of many key reliability and verification questions within this theory, this work has been extended in many ways.

One line of research studies fragments and modifications of this base theory which are decidable. Notably, in 1977, Makanin proved that the satisfiability problem for the quantifier-free theory of word equations is decidable [Makanin, 1977]. In a sequence of papers, Plandowski and co-authors showed that the complexity of this problem is in PSPACE [Plandowski, 2006]. Stronger results have been found where equations are restricted to those where each variable occurs at most twice [Diekert and Robson, 1999] or in which there are at most two variables [Charatonik and Pacholski, 1991; Ilie and Plandowski, 2000;

Dabrowski and Plandowski, 2004]. In the first case, satisfiability is shown to be NP-hard; in the second, polynomial (which was improved further in the case of single variable word equations [Dabrowski and Plandowski, 2002]).

Schulz [Schulz, 1992] extended Makanin's satisfiability algorithm to the class of formulas where each variable in the equations is specified to lie in a given regular set (i.e. a set defined by a regular language). This is a strict generalization of the solution sets of word equations. Further work in [Karhumäki *et al.*, 2000] shows that the class of sets expressible through word equations is incomparable to that of regular sets. Matiyasevich extends Schulz's result to decision problems involving trace monoids and free partially commutative monoids [Diekert *et al.*, 1997; Diekert *et al.*, 1999; Matiyasevich, 1997].

Concurrently, many researchers have looked for the exact boundary between decidability and undecidability. Durnev [Durnev, 1995] and Marchenkov [Marchenkov, 1982] both showed that sentences over word equations is undecidable. Despite decades of effort, however, the satisfiability problem for the quantifier-free theory of word equations and numeric length remains open [Ganesh *et al.*, 2013; Makanin, 1977; Matiyasevich, 2006; Plandowski, 2006].

More recently, Artur Jez presents a technique called re-compression that gives more efficient algorithms for many fragments of theory of word equations [Jeż, 2016]. A related result was shown by Furia [Furia, 2010], wherein he proved that the quantifier-free theory of integer sequences is decidable. The framework he establishes in that paper is closely related to the theory of concatenation and word equations, but weaker than either strings plus numeric length or the theory of arrays due to the inability of the theory of sequences to express facts relating indices directly to elements.

Word equations augmented with additional predicates yield richer structures which are relevant to many applications. In the 1970s, Matiyasevich formulated a connection between string equations augmented with integer coefficients whose integers are taken from the Fibonacci sequence and Diophantine equations [Matiyasevich, 1968; Matiyasevich, 2006]. In particular, he showed that proving undecidability for the satisfiability problem of this theory would suffice to solve Hilbert's Tenth Problem in a novel way.

Though the satisfiability problem of quantifier-free theory of word equations and numeric length remains open, the satisfiability problem for `replace` function is undecidable. Specifically, recursive string functions such as `replace` that are applied to any number of

occurrences of a string (even limited to single-character strings) would make the satisfiability problem undecidable [Buchi and Senger, 1988; Bjørner *et al.*, 2009; Barcelo *et al.*, 2012; Lin and Barceló, 2016].

2.2 Entailment Proving for Automated Verification

We consider the problem of automated verification of imperative programs with emphasis on reasoning about the functional correctness of dynamically manipulated data structures. In this problem domain, pre/post conditions are specified for each function and an invariant is given for each loop before the reasoning system automatically checks if the program code is correct w.r.t. the given annotations.

To explain the relationship between the automated verification and entailment proving, let us start with Hoare Logic [Floyd, 1967; Hoare, 1969], a formal system for reasoning about program correctness. In Hoare Logic, we specify partial correctness of programs using specifications of the form $\{\phi\}$ C $\{\varphi\}$, where C is some code fragment, ϕ is the pre-condition, and φ is the post-condition. Both ϕ and φ are formulas over the program variables in C. The meaning of the triple is as follows: for all program states σ_1, σ_2 such that $\sigma_1 \models \phi$ and executing σ_1 through C derives σ_2, then $\sigma_2 \models \varphi$. For example, the triple $\{x < y\}$ x := x+1 $\{x \leq y\}$ is valid. Note that under this definition, a triple is automatically valid if C is non-terminating or otherwise has undefined behavior.

Automating Hoare Logic is based on generating verification conditions. A verification condition (VC) is a formula Ψ generated automatically from source code and annotated loop invariants. Furthermore, the program obeys specifications if Ψ is valid. In this paradigm, program verification systems first generate VC formulas from source code, and then use theorem prover to check the validity of these formulas [Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Duck *et al.*, 2013; Piskac *et al.*, 2013; Pek *et al.*, 2014; Brotherston *et al.*, 2016]. To illustrate, let us look at the following imperative program:

pre-condition: `even(x)` $\wedge$ `x` $\geq$ `0`
`int add_2(int x) { return x + 2; }`
post-condition: `even(ret)` $\wedge$ `ret` $\geq$ `2`

where `even` predicate is defined as:

$$\texttt{even(x)} \stackrel{def}{=} (\texttt{x} = 0) \vee (\exists \texttt{y} : \texttt{y} = \texttt{x} - 2 \wedge \texttt{even(y)})$$

Suppose the function `add_2` requires the input `x` be even. And we want to assert that the return value of that function application is still an even number. To do this, we need to prove that the following entailment holds:

$$\texttt{even(x)} \wedge \texttt{x} \geq 0 \ \wedge \ \texttt{ret=x+2} \ \models \ \texttt{even(ret)} \wedge \texttt{ret} \geq 2$$

In short, the (safety) properties of programs will be represented using entailments and the correctness of programs will correspond to the validity of such entailments.

For heap manipulating programs, typical correctness properties often require complex combinations of structure, data, and *separation*. To address those properties, Separation Logic [Reynolds, 2002b; Ishtiaq and O'Hearn, 2001; O'Hearn *et al.*, 2001; Reynolds, 2000; O'Hearn and Pym, 1999], an extension of Hoare Logic, is introduced. Separation Logic extends predicate calculus with new logical connectives (namely empty heap (**emp**), singleton heap ($p \mapsto v$), and separating conjunction ($H_1 * H_2$)) such that the structure of assertions reflects the structure of the underlying heap. For example, the pre-condition in the valid Separation Logic triple $\{(x \mapsto \_) * (y \mapsto 2)\}\ [x] := [y] + 1\ \{(x \mapsto 3) * (y \mapsto 2)\}$ represents a heap comprised of two disjoint singleton heaps, indicating that both x and y are allocated and that location y points to the value 2. Here the notation [p] represents pointer dereference. In the post-condition, we have that x points to value 3 as expected. Separation Logic also allows recursively-defined heaps for reasoning over data-structures, such as lists, trees. Compared to Hoare triples, Separation Logic triples have a slightly different meaning regarding memory-safety. A Separation Logic triple $\{\phi\}$ C $\{\varphi\}$ additionally guarantees that any state satisfying ϕ will not cause a memory access violation in C. For example, the triple $\{\mathbf{emp}\}\ [x] := 1\ \{(x \mapsto 1)\}$ is invalid since x is a dangling pointer in any state satisfying the pre-condition.

2.2.1 Unfold-and-Match Techniques

After reducing the problem of automated verification into the problem of entailment proving, we next focus on handling proof obligations of the form $A \models B$. Given an input obligation, a proof system will apply reduction rules to transform that obligation into new obligations

which are also in the form of entailment.

The state-of-the-art proof techniques for user-defined predicates, which are used to represent data structures properties, are Unfold-and-Match techniques [Navarro and Rybalchenko, 2011; Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Piskac *et al.*, 2013; Trinh *et al.*, 2013; Pek *et al.*, 2014]. An entailment proof, using Unfold-and-Match, succeeds when we find a sequence of successive applications of these transformation steps that produce a final formula which is *obviously* provable. This usually means that either (1) there is no recursive predicate in the RHS of the proof obligation and a direct proof can be achieved by consulting a generic SMT solver; or (2) no special consideration is needed on any occurrence of a predicate appearing in the final formula. For example, if $\mathtt{p}(\tilde{u}) \wedge \cdots \models \mathtt{p}(\tilde{v})$ is the formula, then this is obviously provable if $\tilde{u}$ and $\tilde{v}$ were *unifiable* (under an appropriate theory governing the meaning of the expressions $\tilde{u}$ and $\tilde{v}$). In other words, we have performed "formula abstraction" [Madhusudan *et al.*, 2012] by treating the recursively defined term $\mathtt{p}()$ as *uninterpreted*.

Next, we illustrate how Unfold-and-Match techniques work via examples. Below is the proof tree for the simple proof obligation mentioned above. (The proof steps are written in bottom-up order.)

$$\dfrac{\dfrac{\dfrac{}{\texttt{True}}\ (\textsc{obvious})}{\texttt{even(x)} \wedge \texttt{x} \geq 0 \ \wedge\ \texttt{ret=x+2} \ \models\ \texttt{even(x)} \wedge \texttt{x} = \texttt{ret} - 2 \wedge \texttt{ret} \geq 2}\ (\textsc{substitute})}{\texttt{even(x)} \wedge \texttt{x} \geq 0 \ \wedge\ \texttt{ret=x+2} \ \models\ \texttt{even(y)} \wedge \texttt{y} = \texttt{ret} - 2 \wedge \texttt{ret} \geq 2}\ (\textsc{right-unfold})$$
$$\texttt{even(x)} \wedge \texttt{x} \geq 0 \ \wedge\ \texttt{ret=x+2} \ \models\ \texttt{even(ret)} \wedge \texttt{ret} \geq 2$$

Figure 2.1: Proving with Unfold-and-Match

The first step is to unfold the predicate `even(ret)` on the right hand side to obtain $\texttt{even(y)} \wedge \texttt{y} = \texttt{ret} - 2$. (For simplicity, we ignore the existential variable `y`.) By matching/substituting `y` with `x`, we can prove that the entailment holds.

Now let us use another example where we need to reason about heaps. First, let a list segment $\widehat{\texttt{ls(x,y)}}$ denote a portion of the heap (possibly empty if `x=y`) containing an acyclic path from `x` to `y` following the 'points to' relation. Specifically, we have

$$\widehat{\texttt{ls(x,y)}} \stackrel{def}{=} (\texttt{x=y} \wedge \mathbf{emp}) \vee (\texttt{x}\neq\texttt{y} \wedge (\texttt{x} \mapsto \texttt{t}) * \widehat{\texttt{ls(t,y)}})$$

In the above definition, $*$ (from Separation Logic) is the union of disjoint portions of the

heap. Informally, a list segment $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y})$ is an empty heap or an union of two disjoint portions of the heap: a singleton heap $\mathtt{x} \mapsto \mathtt{t}$ and another list segment $\widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$ from $\mathtt{t}$ to $\mathtt{y}$.

$$\begin{array}{l} \texttt{Pre-condition:}\quad \widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \\ \quad \texttt{assume}(\mathtt{x} \neq \mathtt{y}) \\ \quad \mathtt{z} = \mathtt{x}.\mathtt{next} \\ \texttt{Post-condition:}\ \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y}) \end{array}$$

Starting with a list segment from $\mathtt{x}$ to $\mathtt{y}$, we now assume $\mathtt{x} \neq \mathtt{y}$, and $\mathtt{x}$ points to $\mathtt{z}$. We want to assert that we now have a list segment from $\mathtt{z}$ to $\mathtt{y}$. In short, we have to prove the following entailment:

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \ \wedge\ \mathtt{x}{\neq}\mathtt{y} \ \wedge\ (\mathtt{x} \mapsto \mathtt{z}) \ \models\ \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})$$

Below is the proof tree for the above obligation:

$$\dfrac{\dfrac{\dfrac{\texttt{True}}{(\mathtt{x} \mapsto \mathtt{z}) * \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x} \mapsto \mathtt{z}) \models \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})}\ (\text{SUB})}{(\mathtt{x} \mapsto \mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x} \mapsto \mathtt{z}) \models \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})} \qquad \dfrac{\texttt{True}}{\mathtt{x} = \mathtt{y} \wedge \mathbf{emp} \wedge \mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x} \mapsto \mathtt{z}) \models \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})}}{\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x} \mapsto \mathtt{z}) \models \widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})}\ (\text{RU})$$

Figure 2.2: Another Example with Unfold-and-Match

Similarly to the proof in Figure 2.1, the first step is to unfold the predicate $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y})$ on the left hand side to obtain two cases. In the first case, by matching/substituting $\mathtt{t}$ with $\mathtt{z}$, we can prove that the entailment holds. In the second case, since we have a conflict between $\mathtt{x} = \mathtt{y}$ and $\mathtt{x} \neq \mathtt{y}$, the entailment also holds.

Chapter 3

Lazy Reasoning

In this chapter, we introduce our lazy reasoning technique, namely Unfold-and-Consume, for string solving. Before motivating the introduction of this technique, we discuss again the importance of symbolic string solving and the language we need in order to ensure the security of web applications. Next, we present its implementation in a string theory solver of the state-of-the-art SMT solver Z3. Finally, we demonstrate the applicability of this new solver in detecting vulnerabilities in web applications.

3.1 Introduction

Web applications nowadays provide critical services over the Internet and frequently handle sensitive data. Unfortunately, the development is error prone, resulting in applications that are vulnerable to attacks by malicious users. The global accessibility of critical web applications make this an extremely serious problem.

According to the Open Web Application Security Project, or OWASP for short [OWASP, 2013], the most serious web application vulnerabilities include: (#1) Injection flaws (such as SQL injection) and (#3) Cross Site Scripting (XSS) flaws. These two vulnerabilities occur mainly due to inadequate sanitization and inappropriate use of input strings provided by users.

How Important is Symbolic String Solving?

To explain why we need string solving, let us look at *dynamic analysis* which involves testing an application as a closed entity with a set of concrete inputs. Its main disadvantage is of course that it is not a complete method. For example, some program paths may only be executed if certain inputs are passed as parameters to the application, but it is very unlikely that a dynamic analyzer can exhaustively test an application with all possible inputs. For web applications, the problem is even more severe since dynamic analysis needs to take into account not only the value space (i.e., how the execution of control flow paths depends on input values), but also an application's event space (i.e., the possible sequences of user-interface actions). As a result, there is in general an impractical number of execution paths to systematically explore, leading to the "low code coverage" issue of dynamic analysis.

A standard approach to have good or complete coverage is static analysis. However, the problem here is the existence of false positives, arising from an over-approximation of the program's behavior. Recent works to avoid false positives, but still preserve high code coverage, are based on *dynamic symbolic execution* (DSE). Some examples are [Saxena *et al.*, 2010; Bisht *et al.*, 2010; Bisht *et al.*, 2011; Kiezun *et al.*, 2009b; Emmi *et al.*, 2007; Sen *et al.*, 2005; Godefroid *et al.*, 2005; Godefroid *et al.*, 2008; Halfond *et al.*, 2009; Ghosh *et al.*, 2013; Sen *et al.*, 2013; Chaudhuri and Foster, 2010; Bucur *et al.*, 2014; Maras *et al.*, 2013; Wang *et al.*, 2013; Jensen *et al.*, 2013]. These approaches employ both concrete and symbolic execution to automatically and systematically generate tests in order to expose vulnerabilities in web applications. DSE for automated test generation involves instrumenting and *concolically* running a program while collecting path constraints on the inputs. Then it attempts to derive new inputs – using an SMT (Satisfiability Modulo Theories) solver – with the hope to steer next executions toward new program paths. For vulnerability detection, DSE combines the derived path constraints with the specifications for attacks, often given by the security experts, to create queries for the SMT solver.

	Dynamic Analysis	DSE
Code Coverage	Potentially Low	High
False Positives	Low	Low
Executable Paths (EPs)	Unlikely to cover all EPs	Likely to cover all EPs

Table 3.1: DSE as a More Effective Paradigm

In fact, there is a strong connection between an effective vulnerability detection frame-

```
1  ...
2  <html>
3  ...
4  <script>
5  function validateEmail(form) {
6    var email = form["email"].value;
7    var index = email.indexOf("@");
8    var local = email.substr(0, index);
9    var domain = email.substr(index+1);
10
11   if (domain.equals("nus.edu.sg")){
12     var re = new RegExp("^[a-zA-Z][0-9]*$");
13     var test1 = re.test(local);
14     var test2 = local.length == 8;
15     return test1 && test2;
16   }
17   else if (domain.equals("comp.nus.edu.sg"))
18     return local.length >= 4;
19   else
20     return false;
21 }
22 </script>
23 ...
24 <form name="loginForm" action="/Login" onsubmit="return
       validateEmail(this);">
25   Email: <input type="text" name="email" size="64" />
26   <input type="submit" value="Login" />
27 </form>
28 ...
29 </html>
```

Figure 3.1: An Example of Email Address Validation

work and symbolic string solving. As shown in Table 3.1, DSE achieves higher code coverage. However, because not all path executed by DSE are guaranteed to be executable, to avoid false positives we must be able to *decide* if a (symbolic) path constraint is satisfiable or not. Thus a powerful SMT solver, capable of handling symbolic string variables, is the *key* to achieve efficient analyses with high code coverage and low false positives.

To illustrate more clearly how constraint solvers can be helpful in securing web applications, in Fig. 3.1, we present a JavaScript function which is used to validate input email addresses. The user fills the client-side form, by providing an email address to the HTML input element with name `"email"` (and a password, removed for simplicity). When the `Login` button is clicked, the browser invokes the JavaScript validating function `validateEmail`, which is assigned to the `submit` event of the form. This function first fetches the email address supplied by the user from the corresponding `form` field and then checks if the `email`

address is valid. Each student of our department has two email accounts, one from NUS (`nus.edu.sg`), the other from SoC (`comp.nus.edu.sg`). The web page hence accepts both of these two domains. However, these two types of accounts have different formats. While the `local` part of the former is constructed by one alphabetic characters, followed by seven numeric ones, the latter's simply requires at least four characters.

The question is whether this web page is vulnerable to an XSS attack, or to an SQL injection. More specifically, can the following PHP code, with an appropriate instantiation for string variable `$eml`, be executed on the server side, leading to an attack:

```
$eml = $_POST['email'];
$pwd = $_POST['password'];
$stm="SELECT ... where email='$eml' and password='$pwd'";
$result = mysql_query($stm);
```

The answer is *yes* for both of the questions. Now, let us explain the way DSE detects possible vulnerabilities, in comparison with typical dynamic analyses. Since a dynamic analysis is essentially black-box testing, it has no knowledge about the JavaScript code. Thus, it is possible that the dynamic analyzer does not test with email addresses whose domain is `comp.nus.edu.sg`, and subsequently, cannot detect SQL injection and XSS vulnerabilities. In contrast, DSE, which can be seen as white-box testing, enables us to attempt all execution paths by generating three path constraints, corresponding to the three program paths of the `validateEmail` function.

After symbolically executing the program, DSE frameworks such as [Saxena *et al.*, 2010] will combine its results with the specifications for attacks, given by the security experts, to create queries for the constraint solver. The specifications, often come in form of assertions, are some (regular) grammars encoding a set of strings that would constitute an attack against a particular sink. If the constraint solver finds a solution to a query, then this represents an attack that can reach the critical sink and exploit a code injection vulnerability. For example, with the specification to assert if the input email address contains `' OR 1=1--`, we can in fact generate the input

```
' OR 1=1--@comp.nus.edu.sg
```

that leads to an SQL injection. Similarly, a specification for an XSS attack

```
<script>alert('Test')</script>
```

would help us to generate the input email address

```
<script>alert('Test')</script>@comp.nus.edu.sg
```

that can be exploited by attackers.

In summary for this subsection, DSE, presently the state-of-the-art in vulnerability detection, is intimately tied to being able to provide definitive answers for the derived constraint queries. In the case of JavaScript and web applications, since the constraints often concern string variables, symbolic string solving is thus the key to detect vulnerabilities in this class of applications. As the encountered string constraints may be in an undecidable class, it is important to have a solver which returns a definitive answer *often* and in a *timely* manner.

We next describe the main contribution of this work, a new constraint solver S3, which stands for **S**ymbolic **S**tring **S**olver. Our solver makes use of Z3 [De Moura and Bjørner, 2008b], in order to leverage the recent advances in modern SMT solvers.

What Language Do We Need?

We first argue that a pure string language does not suffice to analyze web applications. This is due to the fact that non-string operations (e.g., boolean, arithmetic constraints) are also widely used in web applications. Moreover, their use is often intertwined with string operations, such as in the case of string length — a string-to-integer constraint. Reasoning about strings and non-strings *simultaneously* is thus necessary. In other words, we need to deal with a *multi-sorted* theory which includes, at least, strings and integers.

To amplify this point, let us now state some statistics from a comprehensive study of practical JavaScript applications [Saxena *et al.*, 2010]. Constraints arising from the applications have an average (per benchmark query) of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. The largest fraction are for operations like `indexOf`, `length` (78%). A significant fraction of the operations, including `substring` (5%), `replace` (8%), and `split`, `match` (1%). Of the `match`, `split` and `replace` operations, 31% are based on regular expressions. Operations such as `replace` and `split` give rise to new strings from the original ones, thereby giving rise to constraints involving *multiple* string variables.

To summarize, constraints of interest are either non-strings (e.g., bool-sort, int-sort and

particularly length constraints) or strings such as: string equations, membership predicates and high-level string operations, which are over multiple string variables. It is folklore that query with just basic string equations along with length constraints on the string variables is extremely hard to solve (its decidability is open). Therefore, the validation of any approach can only realistically be done *empirically*.

S3: A Robust and Incremental String Solver

Although there exist solvers that can reason about both string and non-string constraints (e.g., [Saxena *et al.*, 2010; Bjørner *et al.*, 2009; Redelinghuys *et al.*, 2012; Zheng *et al.*, 2013]), they depend on strings being *bounded* in length. Unbounded regular expressions, which can be constructed using Kleene star operation, are not supported. Moreover, the supported high-level operations are only in bounded forms. For example, instead of fully supporting `replace` function, which could mean replacement of all occurrences, existing tools support an operation to replace a fixed number of occurrences in a string.

It may be argued that certain bounds suffice for a class of applications. There is a *more important* reason why the bound dependency is bad: the algorithms that rely on the bounded reasoning are highly *combinatorial* in approach. In other words, the problem at hand is broken down into cases, the number of which is often a large combinatorial combination arising from some given bounds.

Finally, we mention [Alkhalaf *et al.*, 2012a], where there is a real requirement for reasoning about unbounded strings. In verifying client-side input validation functions, a bounded string solver can only find policy violations but it cannot prove the conformance to a given policy. There are certainly some solvers [Hooimeijer and Weimer, 2009; Emmi *et al.*, 2007; Wassermann and Su, 2007; Wassermann and Su, 2008] that can reason about unbounded strings. However, their key weakness is that they cannot handle non-string constraints, particularly length constraints. As shown in the statistics above, missing length constraints (whose appearance is frequent) will lead to many false positives. This clearly is not acceptable.

With regard to all the arguments above, we now conclude this Section with three important features of S3. First, S3 is *expressive* (Section 3.2). Specifically, it is the first to handle unbounded regular expressions in the presence of length constraints, and express precisely high-level string operations, which ultimately enables a more accurate string analysis.

Second, S3 is *robust*. This means that S3 is able to provide definitive answers to a new level, far beyond the state-of-the-art. This in turn means we can detect more vulnerabilities and more bugs. We demonstrate in Section 3.6 with two case studies:

- The first is to compare with Kaluza – the core of Kudzu [Saxena *et al.*, 2010] – a JavaScript symbolic execution framework. We show that S3 is several times faster, and helps detect many more paths that reach the critical sink, that is, paths that are vulnerable.
- The second is to compare with Z3-str [Zheng *et al.*, 2013]. We show that S3 reasons about length constraints much more effectively than Z3-str. This leads to a large increase in applicability to web programs, because this kind of constraints is widely used.

Third, S3 is *efficient*, and one key reason is that it is *incremental*. Our algorithm for string theory is designed in an incremental fashion driven by the try-and-backtrack procedure of the Z3 core (Section 3.4), so that given a set of input constraints, we perform incremental reduction for string variables until the variables are bounded with constant strings/characters. Another technical challenge is how to reason, effectively and efficiently, about the Kleene star and high-level operations such as `replace` (in its most general usage), of which the semantics are by nature recursively defined. Section 3.5 introduces the gist of our proposal, the encodings using recursively-defined functions, on which we can incrementally reason: by lazily *unfolding* them.

3.2 The Constraint Language

We introduce the constraint language of our solver in Fig. 3.2. For simplicity, we only list three primitive types: int, bool and string[1]. The input formula can be of the following forms:

• a boolean expression;

• a comparison operation between two integer or boolean expressions;

• an equation between two string expressions. S3 also supports other common string operations. We list here only important ones;

[1] Z3 supports more primitive types [De Moura and Bjørner, 2008b].

Assertion	::=	**assert** (*(Fml:bool)*)
Fml:bool	::=	*(Term:bool)* | *(Term:bool)* = *(Term:bool)*
	|	*(Term:int)* $\{<, \leq, =, \geq, >\}$ *(Term:int)*
	|	*(Term:str)* = *(Term:str)* | *(Term:str)* $\in$ *(Term:regexpr)*
	|	$\neg$ *(Fml:bool)* | *(Fml:bool)* $\{\wedge, \vee, \Rightarrow\}$ *(Fml:bool)*
Term:bool	::=	*(Var:bool)*
	|	**true**
	|	**false**
	|	**contains**(*(Term:str), (Term:str)*)
Term:int	::=	*(Var:int)*
	|	*Number*
	|	*(Term:int)* $\{+, -, \times, \div\}$ *(Term:int)*
	|	**length**(*(Term:str)*)
	|	**indexOf**(*(Term:str), (Term:str)*)
	|	**search**(*(Term:str), (Term:regexpr)*)
	|	**test**(*(Term:regexpr), (Term:str)*)
Term:str	::=	*ConstString*
	|	*(Var:str)*
	|	*(Term:str)* $\cdot$ *(Term:str)*
	|	**concat**(*(Term:str), (Term:str)*)
	|	**substring**(*(Term:str), (Term:int), (Term:int)*)
	|	**replaceN**(*(Term:str),(Term:regexpr),(Term:str),(Term:int)*)
	|	**replaceAll**(*(Term:str), (Term:regexpr), (Term:str)*)
L:str list	::=	**match**(*(Term:str), (Term:regexpr)*)
	|	**split**(*(Term:str), (Term:regexpr)*)
	|	**exec**(*(Term:regexpr), (Term:str)*)
Term:regexpr	::=	*ConstString*
	|	*(Term:regexpr)**
	|	*(Term:regexpr)* $\cdot$ *(Term:regexpr)*
	|	*(Term:regexpr)* + *(Term:regexpr)*

Figure 3.2: The Grammar of Our Constraint Language

• a membership predicate between a string expression and a regular expression, where an expression can either be a string constant, a variable or their concatenation[2], and regular expressions are constructed from string constants using concatenation ($\cdot$), union ($+$) and Kleene star (*);

• a composite formula constructed using negation and binary connectives, including $\wedge$, $\vee$, $\Rightarrow$.

Z3-str [Zheng *et al.*, 2013] and Kaluza [Saxena *et al.*, 2010] are important existing solvers that can support both string and non-string operations, especially the length con-

[2]We use $x \cdot y$ as a shorter form for **concat**(x, y).

straint. Compared to the constraint syntax of Z3-str, ours can be viewed as an extension with regular expressions, membership predicates, and high-level string operations that often work on regular expressions such as **search**, **replaceAll**[3], **match**, **split**, **test**, **exec**. Our constraint language is also slightly more expressive than Kaluza's since we handle above string operations in its original semantics — unbounded.

In addition, we note that our constraint language, which is necessary to reason about high-level string operations in scripting languages, is beyond the class of context free languages. To illustrate, let us look at the following constraints, in which x can be of any string in the context-sensitive language $\{\ a^n{\cdot}b^n{\cdot}c^n \mid n{\geq}0\ \}$:

$$x = y \cdot z \cdot t \wedge y \in a^\star \wedge z \in b^\star \wedge t \in c^\star \wedge$$
$$\mathbf{length}(y){=}\mathbf{length}(z) \wedge \mathbf{length}(z){=}\mathbf{length}(t)$$

Therefore, existing solvers, which only approximate strings using context free grammars, are not able to reason about the constraints addressed by this work.

Finally, though it is not shown in Fig. 3.2, S3 is able to accommodate most regular expression features in JavaScript via a preprocessing step as done in Kudzu [Saxena *et al.*, 2010]. Examples are (possibly negated) character classes, escaped sequences, repetition operators ({n}/?/⋆/+/) and sub-match extraction using capturing parentheses.

3.3 Motivating Examples

In this Section, we present two simplified examples to position our work against the state-of-the-art.

A JavaScript Program	Generated Constraints	Our Internal Representation

`function validateFields(p1,p2)`
`{`
`  var re1 = /^(ab)*$/;`
`  var re2 = /^(bc)*$/;`
`  var t1 = re1.test(p1);`
`  var t2 = re2.test(p2);`
`  var t3 = p2.length > 0;`
`  return (t1 && t2 && t3)`
`}`
	p1 ∈ ("*ab*")⋆ ∧
	p2 ∈ ("*bc*")⋆ ∧
	length(p2) > 0 ∧
	
	res = p1 · p2 ∧
	nM = "*abababababcc*" ∧
	res = nM
		p1 = **star**("*ab*",n1) ∧
		p2 = **star**("*bc*",n2) ∧
		length(p2) > 0 ∧
		
		res = p1 · p2 ∧
		nM = "*abababababcc*" ∧
		res = nM

Figure 3.3: From a JavaScript Program to the Generated Constraints

In Fig. 3.3 we start with an example of a regular-expression-based input validation

[3]This operation is used to replace all occurrences.

function. The first column is the JavaScript function used to validate the two input fields, namely `p1` and `p2`. This function ensures that `p2` is not an empty string and `p1` (and `p2`) must belong to the regular expressions `re1` (and `re2`) respectively. Given the inputs which have passed the validation function, we want to prove that the output `res`, constructed by concatenating `p1` with `p2`, is different from a specified bad string `nM` = "abababababcc". Ultimately, the above question is reduced to the problem of deciding the satisfiability of the generated constraint formula, presented in the second column of Fig. 3.3. The proof succeeds if the formula is unsatisfiable[4]. This requires reasoning about string equation `res = p1·p2`, membership predicates `p1` $\in$ ("*ab*")* and `p2` $\in$ ("*bc*")*, and length constraint **length**(`p2`) > 0. In short, it becomes a complicated problem involving strings, non-strings and their combinations (e.g., length constraints).

Now, let us discuss how existing solvers would deal with this particular problem. HAMPI [Kiezun *et al.*, 2009a], and other solvers [Christensen *et al.*, 2003; Shannon *et al.*, 2009; Hooimeijer and Weimer, 2010; Veanes *et al.*, 2010; Alkhalaf *et al.*, 2012b; Yu *et al.*, 2010; Hooimeijer and Weimer, 2009; Tateishi *et al.*, 2011; Gange *et al.*, 2013], which work in the string domain only, cannot handle this example. Since they only support string operations, they are not able to handle non-string constraints, and particularly length constraints that are related to both string and non-string domain and cannot be captured in each individual one.

On the other hand, the solvers Kaluza [Saxena *et al.*, 2010], [Bjørner *et al.*, 2009] and Z3-str [Zheng *et al.*, 2013] are in the same category as ours, and can reason about strings and non-strings simultaneously. Since [Bjørner *et al.*, 2009] is similar to Kaluza in many ways, we will just focus on Kaluza here. Kaluza is the string solver used in a JavaScript dynamic test generation framework [Saxena *et al.*, 2010]. To support a wider range of constraint types including integer, boolean and string, it extends both STP [Kiezun *et al.*, 2009a] and HAMPI.

One major drawback of Kaluza is that it requires the lengths of string variables to be known prior to being able to encode them and query the underlying SMT solvers. In particular, before solving string constraints, Kaluza finds a set of satisfying solutions for each string length. For each possible length, it encodes each string variable as an array of bits and then queries a bit-vector solver. Kaluza is unable to reuse the encodings and the

[4]Otherwise, the solver should return satisfying assignments, representing a potential bug/vulnerability of the system.

result of bit-vector solver in previous calls, which induces the overall high cost of repetitive encoding and querying external solvers.

For the example at hand in Fig. 3.3, Kaluza first needs to come up with a set of satisfying solutions for the lengths of `p1` and `p2`, each denoted by a pair $\langle l_1; l_2 \rangle$, where l_1 is the length of `p1` and l_2 is the length of `p2`. In this case, the set of satisfying solutions for the lengths is $\{\langle 0; 14\rangle, \langle 2; 12\rangle, \langle 4; 10\rangle, \langle 6; 8\rangle, \langle 8; 6\rangle, \langle 10; 4\rangle, \langle 12; 2\rangle\}$. For each possible length solution, Kaluza encodes the string variables, and then queries the external bit-vector solver, before finding out that the original set of constraints is unsatisfiable. Overall, Kaluza needs to encode and query bit-vector solver 7 times. Let us not have the impression that, in general, the number of satisfying solutions for the string lengths should be of this linear complexity. In fact, practical applications involve many string variables, Kaluza approach, i.e., generate-and-test, would easily suffer from a *combinatorial* explosion.

Z3-str [Zheng *et al.*, 2013] cannot handle regular expressions, thus also cannot handle this example. However, it can be considered the first SMT-based string solver. Instead of relying on other theories, it builds a string theory for itself and allows this string theory to be plugged into a modern and powerful solver – Z3 [De Moura and Bjørner, 2008b]. Thus an important contribution of Z3-str is that string and non-string constraints are now solved simultaneously, in an *incremental* manner.

Inspired by Z3-str's design, our target is to build a string theory that can interact with other theories via Z3. Nevertheless, we want to support a powerful input language, which is especially demanded for testing and analysis of practical web applications. There are two key technical challenges: (1) how to *incrementally* handle the Kleene star, which is the heart of the issue in reasoning about regular expressions; (2) how to *incrementally* handle high-level string operation such as `replace`, whose semantics is most naturally defined by recursive rules. Our solution therefore is to employ, in our string theory, recursively defined functions whose semantics will be *lazily* unfolded during the process of incremental solving. Such approach resembles the constrain-and-generate technique (to contrast with generate-and-test) in the literature of constraint solving.

We elaborate later with a technical description in Section 3.5. But now let us give some intuitions on how we approach this example. Internally, we represent membership of regular expression as equation involving a *symbolic representation* of the Kleene star. In particular, `p1`$\in$(“*ab*”)* is represented as **`p1=star(`**“*ab*”,`n1)` and similarly `p2`$\in$(“*bc*”)* is represented as

`p2=star(`"*bc*"`,n2)`. By rewriting, we would derive the following equation:

$$\texttt{star(}\text{“}ab\text{”}\texttt{,n1)} \cdot \texttt{star(}\text{“}bc\text{”}\texttt{,n2)} = \text{“}abababababcc\text{”}$$

Since the length of `p2` is positive and the RHS is a constant string, this would force the *unfolding* of expression **star**("*bc*",`n2`) to **star**("*bc*",`n2-1`) · "*bc*". A conflict is then derived since the LHS string ends with "*bc*" while the RHS string ends with "*cc*". Our system then can conclude that the input formula is UNSAT.

$$x = x_1 \cdot x_2 \ \wedge\ z = y \cdot z_3 \ \wedge\ y = z_1 \cdot z_2 \ \wedge\ z_2 = \text{“}\_\text{”} \ \wedge\ l_1 = \mathbf{length}(x_1) \ \wedge\ l_2 = \mathbf{length}(z_1) \ \wedge\ l_1 = l_2 + 1 \ \wedge\ x = z \ \wedge\ \mathbf{indexOf}(y, \text{“}a\text{”}) = 3 \ \wedge\ \mathbf{indexOf}(x1, \text{“}a\text{”}) = 4$$

Figure 3.4: A Frequent Constraint Pattern

Now let us dissect Z3-str more carefully. Fig. 3.4 presents an input example for Z3-str, a pattern which is commonly found in many benchmarks extracted from the comprehensive set of JavaScript applications of [Saxena *et al.*, 2010] (e.g. big2). Starting with the fact that z_2 is a constant string of one character, Z3-str is able to deduce that z_2 is of length 1. This constraint will be fed into the arithmetic theory. Similarly, the arithmetic theory would receive the information that y's length is the sum of z_1's length and z_2's length. Since, from the input, the length of x_1 equals to the length of z_1 plus 1, the arithmetic theory can deduce that x_1 and y are of the same length. However, this information will *never* be passed back to the string theory.

As discussed in [Zheng *et al.*, 2013], the current design of Z3 enforces that the plug-in theory, namely Z3-str, to be *disjoint* from Z3's arithmetic theory. Being a plug-in, however, means there is supervisory control over Z3-str which can feed length information to the arithmetic theory so that early conflicts can be detected and exploited. But, importantly, partial information derived by the arithmetic theory will not be fed back to Z3-str. This is the source of Z3-str's inefficiency in many cases.

Returning to the example, if the information that x_1 and y are of the same length is propagated back to the string theory, together with the fact that x_1 and y are prefixes of the two equal strings x and z, our string theory can derive that x_1 and y are equal, therefore proceed the search much more efficiently. In Section 3.4.2, we discuss our new design in order to overcome this drawback, therefore even when restricted to the same input language as of Z3-str, our tool, S3, does advance the concept of incremental solving to the next level.

3.4 Design of S3

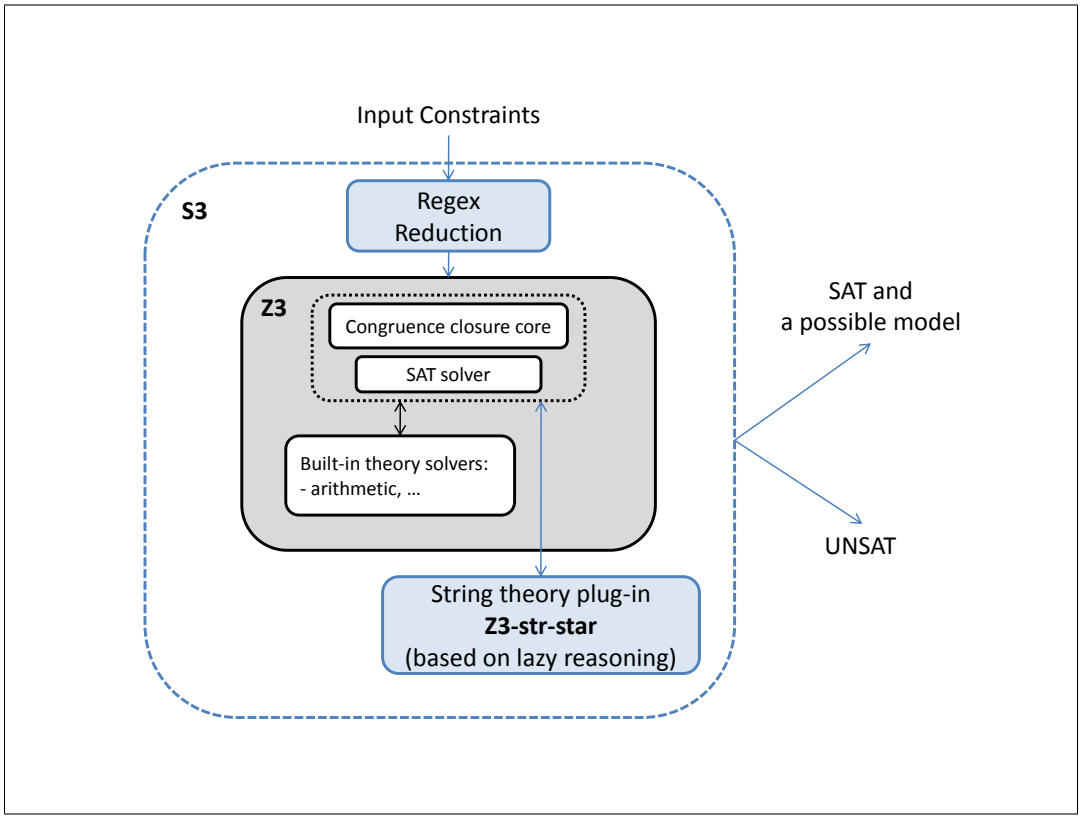


Figure 3.5: The Design of S3

Here we present the design of S3. This design is inspired by Z3-str [Zheng *et al.*, 2013], and thus inherits its two main advantages. First, we support the primitive type of string so that there is no need to convert strings to other representations, e.g., bit-vectors. As a result, we can support string variables whose lengths can be *unknown*, especially in the context of static analysis. Second, we leverage the power of Z3 in dealing with multiple theories, and this ultimately leads to the capability of reasoning on string and non-string constraints simultaneously and efficiently. We first give an overview of Z3-str, focusing on how it interacts with the core of Z3. Later we describe our design of S3, along with the improvement of the corresponding component Z3-str-star over Z3-str.

3.4.1 Overview of Z3-str

Z3-str acts as a plug-in string theory for a SMT solver Z3 [De Moura and Bjørner, 2008b]. The architecture of Z3 is shown in the shaded box of Fig. 3.5. Its core component consists of the following modules: the congruence closure engine, a SAT solver-based DPLL layer, and several built-in theory solvers, such as integer linear arithmetic, bit-vectors, etc. The congruence closure engine can detect equivalent terms and then classify them into different equivalence classes, which are shared among all built-in theory solvers. The SAT-based DPLL layer is responsible for handling the boolean structure of the input formula.

assert $((e_1 \vee e_2) \wedge e_3 \wedge e_4)$	
$e_1 : x =$ "abc" $\cdot\, m$	$e_2 : x =$ "$efgh$"
$e_3 : y =$ "efg" $\cdot\, n$	$e_4 : x = y$

Consider the assertion above. The core component cannot interpret the string operations; instead it treats them as four independent boolean variables (e_1, e_2, e_3 and e_4) and tries to assign boolean values to them. We now walk through the process of how Z3's core component and the string theory solver interact.

	Fact added	Eq-class	Reduction/Action
1	y="efg"$\cdot n$	{y,"efg"$\cdot n$}	
2	x=y	{x,y} {y,"efg"$\cdot n$}	
3	x="abc"$\cdot m$	{x,"abc"$\cdot m$, y,"efg"$\cdot n$}	• conflict detected • backtrack and remove facts • try another option for e_1
4	x="$efgh$"	{x,"$efgh$", y,"efg"$\cdot n$}	"$efgh$"="efg"$\cdot n \Rightarrow n$="h"
SAT solution: $x =$ "$efgh$", $y =$ "$efgh$", $n =$ "h"			

Table 3.2: How Z3-str Interacts with Z3 and Its Backtracking

In Table 3.2, initially there is no fact. The core starts by setting e_3 and e_4 to `true` and reaches step 3. Without loss of generality, assume the core component first tries `true` for e_1. Beware that the core can detect functionally equivalent terms, based on the theory of uninterpreted functions. Hence, it puts {x, y, "abc" $\cdot\, m$, "efg" $\cdot\, n$} into one equivalence class and notifies the string theory plug-in. We note that the plug-in string theory Z3-str can only know about the equivalent terms that belong to its theory. As a side remark, if we have an equation $\mathbf{length}(x) = 4$, then Z3-str is not aware of the fact that $\mathbf{length}(x)$ is equal to 4. However, if e_2 were set to `true`, Z3-str would know that x is equivalent to

a constant string of length 4. Therefore, it can deduce that **length**(x) is equal to 4, thus subsequently passing this information to the arithmetic theory.

Back to the example, with the above equivalence class at step 3, Z3-str detects a conflict and then informs the core component about the new finding through an axiom $e_3 \wedge e_4 \rightarrow \neg e_1$. With this new axiom, the core component backtracks and tries `false` for e_1. When the core component backtracks, it discards the relevant fact and any insertions into equivalence classes as the consequence of the fact. The core then derives that e_2 must be `true` and this assignment is performed in step 4. Based on the concatenation semantics, Z3-str can infer that n must be "h". This new finding is formulated by introducing a new boolean variable e_5 representing $n =$ "h" and an axiom "$efgh$" = "efg" $\cdot\, n \Rightarrow e_5$, which is sent back to the core. From the existing facts and the new axiom, the core component derives e_5 is `true`. After all boolean expressions have been assigned consistently and Z3-str can find the satisfying values for string variables x, y, and n, the search procedure terminates.

3.4.2 Improvement of Z3-str-$\star$ over Z3-str

Z3-str-star (or Z3-str-$\star$ for short), a component of our tool, is responsible for solving equations between string expression and recursively-defined functions. It can be viewed as a *significant extension* of Z3-str with the support of recursively-defined functions, introduced to facilitate representing and reasoning about the Kleene star and commonly used high-level string operations.

As mentioned before, in its current implementation, Z3-str does not know about equivalent terms that belong to other theories, especially the arithmetic theory. Another important improvement of Z3-str-$\star$ (over Z3-str) is its direct interactions with the Z3 core, to query about the equivalence classes among multiple theories. More specifically, it asks Z3 core two following questions:

- Is a string length "ground" with a non-negative constant?
- What is the relationship $(=, <, >, \leq, \geq)$ between different length variables?

To answer these questions, we extend Z3 API so that Z3-str-$\star$ can interact with the congruence closure core, similarly to other built-in theory solvers. Moreover, the newly introduced API methods also help us to query about other inequality relationship, if necessary. Answers to these questions ultimately allow us to propagate the information of string lengths

to string theory solver so that string and non-string constraints can be simultaneously reasoned about. In short, this gives us a truly incremental solver for strings and non-strings. We will revisit this side contribution in our experimental evaluation – Section 3.6.

3.5 Algorithm

3.5.1 Top-level Algorithm

S3 finds a list of string assignments that satisfies the input formula or decides that no satisfying assignment exists. Algorithm 1 summarizes its top level algorithm.

Input: `F` : *Formula*
Output: (IsSat : *bool*, Solutions : *(variable, string) list*)
`reduced_F` $\leftarrow$ **reduce**(`F`);
$\bigvee_i^n$ disjunct$_i$ $\leftarrow$ **normalize_to_DNF**(`reduced_F`);
for $i = 1$ **to** n **do**
  (*Res, Sols*) $\leftarrow$ Z3-str-$\star$(disjunct$_i$);
  if *Res* = *SAT* **then**
    return *(true, Sols)*;
  end
end
return *(false, [])*;

Algorithm 1: Top-level Algorithm

Given an input formula `F`, S3 recursively reduces `F` into new formula `reduced_F`, which may contain equations (among string expressions and recursive functions such as **star**) and length constraints. Here we only take into consideration the string and length constraints, non-string constraints will be unchanged unless otherwise stated. Reduction rules may result in a disjunctive formula. Thus, the next step is to normalize `reduced_F` into disjunctive normal form (DNF). To decide the satisfiability of each disjunct, we extend Z3-str [Zheng *et al.*, 2013] to support recursive functions. In particular, we use the recursive function **star** to represent the Kleene star. For presentation purpose, we first only discuss how to handle the **star** function, calling our extended component Z3-str-$\star$. Similar treatment for high-level operations such as **replaceAll** will be elaborated later. If Z3-str-$\star$ finds a satisfiable disjunct, it stops and returns the corresponding satisfying assignments. Otherwise, it decides that no such assignment exists.

3.5.2 Reduction of Regular Expressions

Rule	Reduction
[**CONST**]	$e \in s \quad \rightarrow \quad e{=}s$
[**UNION**]	$e \in r_1{+}r_2 \quad \rightarrow \quad e \in r_1 \vee e \in r_2$
[**CONCAT**]	$e \in r_1{\cdot}r_2 \quad \rightarrow \quad e{=}e_1{\cdot}e_2 \wedge \bigwedge_{i=1}^{2} e_i \in r_i$
[**STAR**]	$e \in r^{\star} \quad \rightarrow \quad e \stackrel{\vee}{=} \mathbf{star}(r,\ n)$

Table 3.3: Reduction Rules

Given an input constraint formula, we first reduce membership predicates into equations among string expressions and **star** function. The reduction rules are summarized in Table 3.3. Our aim is to obtain a list of new constraints of the form that can be solved *incrementally* by Z3-str-$\star$ — equations among string expressions and recursively-defined functions, along with length constraints.

These rules deal with constraints checking if a string expression e (LHS) is in a regular expression (RHS). If the RHS is merely a string constant, rule [**CONST**] will convert such membership constraint into an equality. The next two rules handle the case when the RHS is constructed by union and concatenation operations. While rule [**UNION**] ensures that the LHS expression e is a member of one of the RHS sub-expressions (of the union), rule [**CONCAT**] splits e into two fresh string variables, namely e_1 and e_2, and checks that they satisfy the condition $e_1 \in r_1 \wedge e_2 \in r_2$ conjunctively.

The RHS regular expression can also be formed by repeating r zero or more times (Kleene star). Rule [**STAR**] encodes such constraint as an equation, where the LHS is a string expression and the RHS is a symbolic representation for a family of strings generated by the Kleene star. The fresh (symbolic) integer variable n indicates the frequency where r is repeated. This symbolic variable is used to:

- Distinguish different **star** functions, which have the same base regular expression (e.g. r).
- Guide the on-demand unfolding in the recursively-defined functions such as **star** or **replaceAll** (which will be discussed later).
- Interact with the Arithmetic Solver module in Z3.

When r is a constant string and n is a concrete value, the $\stackrel{\vee}{=}$ operator is interpreted as equality operator $=$. For convenience, we overload $\stackrel{\vee}{=}$ with the $=$ notation.

In short, after the reduction of regular expressions, we have equations among string expressions and recursively-defined **star** functions, along with length constraints. Z3-str-⋆ is then responsible for solving them.

3.5.3 star Functions

Z3-str-⋆ extends Z3-str [Zheng *et al.*, 2013] with the support for handling **star** functions. The internal language is extended with the following:

$$\begin{array}{rcl} \textit{Term:str} & ::= & \textit{ConstString} \\ & | & \ldots \\ & | & \textbf{star}(\textit{Term:regexpr}, \textit{Term:int}) \end{array}$$

Like Z3-str, Z3-str-⋆ also works as a plug-in of Z3. It is notified by the Z3 core component when a string equation is asserted as part of the try-and-backtrack process. In particular, the core component invokes a callback function in the plug-in, providing the abstract syntax tree (AST) of the equation as an input parameter. The callback function inspects the AST, and if it involves string operations, the function tries to reduce AST to a simpler abstract syntax tree, say AST'. The reduction is conveyed to the core component by adding an axiom with the form of $AST \Rightarrow AST'$. Recall that since the core component does not understand the string domain, it treats both AST and AST' as independent boolean variables. Because AST has been assigned a `true` value, with the new axiom, the core will assign `true` to AST' as well, which is a new fact, and in turn triggers further plug-in processing. Thus, to act as a plug-in, we need to provide reduction rules for each callback from Z3.

We list selected reduction rules in Table 3.4. There are 3 cases of interest related to **star** functions:

- when **star** appears in one side of an equation,
- when **star** appears in both side of an equation and
- when **star** can be used to concretize other concatenations based on its concrete string value.

The gist of our reduction rules is to make use of the semantics of **star** functions (or their previous forms – regular expressions with Kleene star). In fact, with a membership con-

straint such as $\mathtt{x} \in (\text{“}ab\text{”})^\star$, we can directly make use Z3-str to generate the possible string assignments for $\mathtt{x}$, then checking membership is straight-forward since $\mathtt{x}$ is already ground. However, this naive approach is likely to be inefficient. Sometimes, it may be worse than Kaluza's approach, where the lengths can be used to refine the string constraints. To deal with **star** functions effectively and efficiently, we propose to reduce it *lazily* and only *on demand*. We call that technique *"unfold and consume"*. The basic principle is to lazily unfold its semantics, until we find a matching between constant string segments in the two sides of an equation. At that time, we can easily to choose either consume these constants (of course with the capability of backtracking), or to find a conflict between *unmatchable* constants in the two sides.

Incremental Solving for star Functions

In Table 3.4 we introduce four auxiliary functions: $csm\_hd(s, r)$, $csm\_tl(s, r)$, $\overset{r}{csm}\_hd(r_2, r_1)$, and $csm\_all(s, r)$. The first one takes a constant string and a regular expression, and returns a list of strings s_i such that: $s \in r{\cdot}s_i$. Intuitively, this function aims to consume the prefix of s matching r. Similarly, while the second, $csm\_tl(s, r)$, consumes the suffix of s matching r, the third one applies to two regular expressions instead. Lastly, $csm\_all(s, r)$ checks if s can be consumed completely by matching it with r.

Now, let us have a look at reductions rules in Table 3.4. The rule [**CON**$-\star$] says about the case when $\mathbf{star}(r, n)$ equals to some constant string s. As we explained above, method $csm\_all(s, r)$ is used to decide whether s can be a member of $r^\star$. If yes (the second case), we can update other string expressions that contain $\mathbf{star}(r, n)$. Otherwise (the first case), it is a conflict and Z3 core component will need to backtrack. Note that, in Table 3.4, all E_1, E_2 and E_3 are concatenations among string expressions and **star** functions.

The rules [**HD**$-\star$] and [**HT**$-\star$] are to handle the case when there is a matching between **star** and a constant string. In the former, the matching is at the beginning of the LHS; while the latter is a special case of it, where the matchings occur at both ends. These two rules will be elaborated more in the next example. Similarly, we have the rule [**TL**$-\star$] for the matching at the end of the LHS. We have $[s_i]{=}csm\_hd(s_1, r_1)$ and $[s_j]{=}csm\_tl(s_2, r_2)$ in rule [**HT**$-\star$], $[s_i]{=}csm\_hd(s, r)$ in rule [**HD**$-\star$], and $[s_i]{=}csm\_tl(s, r)$ in rule [**TL**$-\star$].

The rule [**HD**$-\star-\star$] ([**TL**$-\star-\star$], [**HT**$-\star-\star$]) is applied when there are two **star** function at the beginning (end or both) of each side of the equation. In the rule [**HD**$-\star-\star$], we assume

Rule	Reduction		
[CON−⋆]	$\mathbf{star}(r,n){=}s$	$\Rightarrow$	$\neg\mathbf{star}(r,n){=}s$
	$\mathbf{star}(r,n){=}s \wedge (E_1{\cdot}\mathbf{star}(r,n){\cdot}E_2{=}E_3)$	$\Rightarrow$	$E_1{\cdot}s{\cdot}E_2{=}E_3$
[HT−⋆]	$\mathbf{star}(r_1,n_1){\cdot}E_1{\cdot}\mathbf{star}(r_2,n_2){=}s_1{\cdot}E_2{\cdot}s_2$	$\Rightarrow$	$(E_1{=}s_1{\cdot}E_2{\cdot}s_2 \wedge n_1{=}0 \wedge n_2{=}0) \vee (\bigvee_{i=1}^{k} \mathbf{star}(r_1,n_1{-}1){\cdot}E_1{=}s_i{\cdot}E_2{\cdot}s_2 \wedge n_2{=}0) \vee (\bigvee_{j=1}^{l} E_1{\cdot}\mathbf{star}(r_2,n_2{-}1){=}s_1{\cdot}E_2{\cdot}s_j \wedge n_1{=}0) \vee \bigvee_{i,j}^{k,l} \mathbf{star}(r_1,n_1{-}1){\cdot}E_1{\cdot}\mathbf{star}(r_2,n_2{-}1){=}s_i{\cdot}E_2{\cdot}s_j$
[HD−⋆]	$\mathbf{star}(r,n){\cdot}E_1{=}s{\cdot}E_2$	$\Rightarrow$	$(E_1{=}s{\cdot}E_2 \wedge n{=}0) \vee \bigvee_{i=1}^{k} \mathbf{star}(r,n{-}1){\cdot}E_1{=}s_i{\cdot}E_2$
[TL−⋆]	$E_1{\cdot}\mathbf{star}(r,n){=}E_2{\cdot}s$	$\Rightarrow$	$(E_1{=}E_2{\cdot}s \wedge n{=}0) \vee \bigvee_{i=1}^{k} E_1{\cdot}\mathbf{star}(r,n{-}1){=}E_2{\cdot}s_i$
[HT−⋆−⋆]	$\mathbf{star}(r_1,n_1){\cdot}E_1{\cdot}\mathbf{star}(r_3,n_3){=}\mathbf{star}(r_2,n_2){\cdot}E_2{\cdot}\mathbf{star}(r_4,n_4)$	$\Rightarrow$	$(n_2{=}0 \wedge n_4{=}0 \wedge \mathbf{star}(r_1,n_1){\cdot}E_1{\cdot}\mathbf{star}(r_3,n_3){=}E_2) \vee (n_2{=}0 \wedge \mathbf{star}(r_1,n_1){\cdot}E_1{\cdot}\mathbf{star}(r_3,n_3){=}E_2{\cdot}\mathbf{star}(r_4,n_4)) \vee (n_4{=}0 \wedge \mathbf{star}(r_1,n_1){\cdot}E_1{\cdot}\mathbf{star}(r_3,n_3){=}\mathbf{star}(r_2,n_2){\cdot}E_2) \vee \bigvee_{i,j}^{k,l} \mathbf{star}(r_1,n_1{-}1){\cdot}E_1{\cdot}\mathbf{star}(r_3,n_3{-}1){=} s_i{\cdot}\mathbf{star}(r_2,n_2{-}1){\cdot}E_2{\cdot}\mathbf{star}(r_4,n_4{-}1){\cdot}s_j$
[HD−⋆−⋆]	$\mathbf{star}(r_1,n_1){\cdot}E_1{=}\mathbf{star}(r_2,n_2){\cdot}E_2$	$\Rightarrow$	$(E_1{=}E_2 \wedge n_1{=}0 \wedge n_2{=}0) \vee (\mathbf{star}(r_1,n_1){\cdot}E_1{=}E_2 \wedge n_2{=}0) \vee (E_1{=}\mathbf{star}(r_2,n_2){\cdot}E_2 \wedge n_1{=}0) \vee \bigvee_{i=1}^{k} \mathbf{star}(r_1,n_1{-}1){\cdot}E_1{=}s_i{\cdot}\mathbf{star}(r_2,n_2{-}1){\cdot}E_2$
[TL−⋆−⋆]	$E_1{\cdot}\mathbf{star}(r_1,n_1){=}E_2{\cdot}\mathbf{star}(r_2,n_2)$	$\Rightarrow$	$(E_1{=}E_2 \wedge n_1{=}0 \wedge n_2{=}0) \vee (E_1{\cdot}\mathbf{star}(r_1,n_1){=}E_2 \wedge n_2{=}0) \vee (E_1{=}E_2{\cdot}\mathbf{star}(r_2,n_2) \wedge n_1{=}0) \vee \bigvee_{i=1}^{k} E_1{\cdot}\mathbf{star}(r_1,n_1{-}1){=}E_2{\cdot}\mathbf{star}(r_2,n_2{-}1){\cdot}s_i$
[REP−⋆]	$x{=}E \wedge (E_1{\cdot}x{\cdot}E_2)$	$\Rightarrow$	$E_1{\cdot}E{\cdot}E_2$

Table 3.4: Selected reduction rules for **star** function

that r_1 cannot be consumed by r_2 so that we only need the auxiliary function $\overset{r}{csm}\_hd(r_2,r_1)$. We have $[s_i]{=}\overset{r}{csm}\_hd(r_3,r_1)$ and $[s_j]{=}\overset{r}{csm}\_tl(r_4,r_2)$ in rule [HT−⋆−⋆], $[s_i]{=}\overset{r}{csm}\_hd(r_2,r_1)$ in rule [HD−⋆−⋆], and $[s_i]{=}\overset{r}{csm}\_tl(r_2,r_1)$ in rule [TL−⋆−⋆].

The last rule [REP−⋆] aims to replace all string variables by their aliases, which are a concatenation among constant strings and **star** functions.

To illustrate how these rules are applied, in Table 3.5, we present running steps for solving the example in Fig. 3.3. Z3 core continually sends the assignments to Z3-str-⋆ (via its call back function) from step 1 to step 5. At the same time, Z3 also maintains functionally equivalent terms in their equivalence classes. From step 1 to step 4, we apply the rule [REP−⋆] repetitively to replace a string variable by a constant string, a **star** function or their concatenation (shown in column 4, step 1-4). In step 5, we apply a specialized version of rule [HT−⋆], where we also make use of constraints on variable n_1 and n_2. More

Step	Fact added	Eq-class	Reduction/Action
1	nM = "*abababababcc*"	{"*abababababcc*", res, nM, p1 · p2}	[REP−⋆]: res = "*abababababcc*"
2	p1 = **star**("*ab*",n1)	{p1, **star**("*ab*",n1)}	[REP−⋆]: res = **star**("*ab*",n1) · p2
3	p2 = **star**("*bc*",n2)	{p2, **star**("*bc*",n2)}	[REP−⋆]: res = **star**("*ab*",n1) · **star**("*bc*",n2)
4	res= **star**("*ab*",n1)·**star**("*bc*",n2)	{"*abababababcc*", res, nM, **star**("*ab*",n1)· **star**("*bc*",n2), p1 · p2}	[REP−⋆]:**star**("*ab*",n1)·**star**("*bc*",n2)= "*abababababcc*"
5	**star**("*ab*",n1) · **star**("*bc*",n2) = "*abababababcc*"	{"*abababababcc*", res, nM, **star**("*ab*",n1)· **star**("*bc*",n2), p1 · p2}	**star**("*ab*",n1)·**star**("*bc*",n2) ="*abababababcc*"⇒ ¬**star**("*ab*",n1)·**star**("*bc*",n2) ="*abababababcc*"
UNSAT			

Table 3.5: A Solving Procedure for the Motivating Example in Fig. 3.3

specifically, for this running example, we are able to force the unfolding of **star**("*bc*",n2) so that we can find a conflict between "*bc*" and "*cc*". Finally, we give back the new axiom (in column 4, step 5) to Z3 so that Z3 can conclude the input formula is UNSAT.

3.5.4 String Operations

Typically, the semantics of string operations such as **replaceAll**, **match**, **split**, **test**, **exec**, are recursively defined. As such, it is natural for us to interpret them as recursively-defined functions, similarly to our handling of **star** functions. In this Subsection, we only give the details of reduction for **replaceAll**. Other operations can be treated in a similar manner.

As stated earlier, we aim to support the most general usage of `replace` function – replacing *all* occurrences. In practice, there is also another version (e.g. in PHP) which allows users to specify the maximum number of occurrences to be replaced. We call it **replaceN**, to distinguish the two versions. In fact, **replaceN** is already supported by existing solvers, e.g., Kaluza. The typical treatment is to model the input parameter as a concatenation of N parts, and then apply one replacement to each part. However, this technique cannot be generalized to address **replaceAll**, since we do not know such an N beforehand. Here we propose to model both **replaceAll** and **replaceN**, again, using recursively-defined functions. In fact, restricting to **replaceN** alone, our approach will be more efficient than Kaluza's. This efficiency comes from the superiority of incremental solving (via constrain-and-generate approach) over generate-and-test approach.

Since **replaceN** is a special case of **replaceAll**, we focus on discussing only the latter. Table 3.6 shows that R=**replaceAll**(S, r, T) belongs to one of two possible cases:

Operations	Reduction Rules
$\mathtt{I}{=}\mathbf{search}(\mathtt{S},\mathtt{r})$	$(\mathtt{I}{<}0 \wedge \neg(\mathtt{S} \in (.^\star)\cdot\mathtt{r}\cdot(.^\star))) \vee (\mathtt{I}{\geq}0 \wedge \mathtt{S}{=}\mathtt{U}\cdot\mathtt{M1}\cdot\mathtt{M2}\cdot\mathtt{R} \wedge \mathtt{M1}\cdot\mathtt{M2} \in \mathtt{r} \wedge \mathbf{length}(\mathtt{U}){=}\mathtt{I} \wedge \mathbf{length}(\mathtt{M2}){=}1 \wedge \neg(\mathtt{U}\cdot\mathtt{M1} \in (.^\star)\cdot\mathtt{r}\cdot(.^\star)))$
$\mathtt{R}{=}\mathbf{replaceAll}(\mathtt{S},\mathtt{r},\mathtt{T})$	$\mathtt{I}{=}\mathbf{search}(\mathtt{S},\mathtt{r}) \wedge ((\mathtt{I}{<}0 \wedge \mathtt{R}{=}\mathtt{S}) \vee (\mathtt{I}{\geq}0 \wedge \mathtt{S}{=}\mathtt{U}\cdot\mathtt{M}\cdot\mathtt{S}_1 \wedge \mathtt{R}{=}\mathtt{U}\cdot\mathtt{T}\cdot\mathtt{R}_1 \wedge \mathtt{M} \in \mathtt{r} \wedge \mathbf{length}(\mathtt{U}){=}\mathtt{I} \wedge \mathtt{R}_1{=}\mathbf{replaceAll}(\mathtt{S}_1,\mathtt{r},\mathtt{T})))$

Table 3.6: Reduction Rules for **search** and **replaceAll**

- the recursive case, when we find a substring $\mathtt{M}$, that matches regular expression $\mathtt{r}$, at an index $\mathtt{I}$. We then can replace $\mathtt{M}$ by $\mathtt{T}$ and continue to apply **replaceAll** function on the remaining part $\mathtt{S}_1$ until we reach the base case.

- the base case, when we cannot find any substring that satisfies such condition. The resulting string $\mathtt{R}$ is then the same as the input string $\mathtt{S}$.

The **replaceAll** function will use **search** function to find the index of substring $\mathtt{M}{=}\mathtt{M}_1\cdot\mathtt{M}_2$ in $\mathtt{S}$. Specifically, this auxiliary function takes as input a symbolic string input $\mathtt{S}$, a regular expression $\mathtt{r}$, and returns the starting index $\mathtt{I}$ of a substring in $\mathtt{S}$ that matches $\mathtt{r}$. If there exists no such substring, it returns a negative number. Otherwise, it returns the index of the substring $\mathtt{M}$ that satisfies the condition.

Rule	Reduction			Condition
[RED−1]	$\mathbf{replaceAll}(s\cdot R, r, T){=}U$	$\Rightarrow$	$V\cdot\mathbf{replaceAll}(t\cdot R, r, T){=}U$	$(V,t){=}rep(s,r,T)$
[RED−2]	$\mathbf{replaceAll}(\mathbf{star}(s,n)\cdot R, r, T){=}U$	$\Rightarrow$	$V\cdot\mathbf{replaceAll}(t\cdot R, r, T){=}U$	$(V,t){=}rep(\mathbf{star}(s,n),r,T)$

Table 3.7: Reduction Rules for **replaceAll** Functions

We remark that the second parameter of **replaceAll** function cannot be a variable since in such case, the behavior of this function is undefined. Naively, we can keep unfolding recursively-defined function **replaceAll**, until we can decide if the current formula is satisfiable or not. However, we provide reduction rules (unfolding on demand) for them instead. For presentation purpose, Table 3.7 lists only two reduction rules for the case when the prefix of the first parameter S is known[5]. In rule [RED−1], the prefix of S is a constant string s, while it is $\mathbf{star}(s,n)$ in rule [RED−2]. In both cases, since the prefix is already known, we are able to apply the replacement on the part s ($\mathbf{star}(s,n)$ in the other case) via auxiliary

[5]Other rules related to the second, the third parameter, the result and their combinations are constructed similarly.

function rep. In rule [**RED**−**1**], suppose that S is composed by s and R, function $rep(s, r, T)$ replaces all occurrences in s, matching the regular expression r, by T. It then returns the pair (V, t) such that **replaceAll**(s, r, T)=$V \cdot t$, where t is guaranteed to be the longest suffix of s that must be examined together with R in the next step **replaceAll**$(t \cdot R, r, T)$. The application of rep for the case **star**(s, n) is similar to s except that V is parameterized by n. Now, we illustrate how this auxiliary function can be applied via two examples. In the first example:

$$\textbf{replaceAll}(\text{“}abcd\text{”} \cdot R, \text{“}ab\text{”}, T) = U$$

the $rep(\text{“}abcd\text{”}, \text{“}ab\text{”}, T)$ method will return $(T \cdot \text{“}cd\text{”}, \text{“”})$. In the second one:

$$\textbf{replaceAll}(\text{“}abcd\text{”} \cdot R, (\text{“}ab\text{”} + \text{“}de\text{”}), T) = U$$

it will return $(T \cdot \text{“}c\text{”}, \text{“}d\text{”})$ since it is possible that R starts with character ‘e’.

Length constraints. We have inherited rules from Z3-str, to infer length constraints such as X=$Y \rightarrow$ **length**(X)=**length**(Y). Importantly, the unfolding of recursive functions (**star**, **replaceAll**, etc.) would incrementally expose more concrete (sub)strings and therefore the interactions from Z3-str-$\star$ to the Arithmetic Solver module in Z3 also happen incrementally.

In addition, as stated in Sec. 3.4.2, the length constraints, in the feedback from the Arithmetic Solver module, can also be used to prune the search space in string theory component, Z3-str-$\star$. For example, when the Arithmetic Solver module can deduce concrete values for length variables, Z3-str-$\star$ will be able to make use of such information.

3.6 Evaluation

In our experimental evaluation, we conduct case studies to compare S3 with state-of-the-art string solvers. All experiments are run on an 3.2GHz machine with 8GB memory.

In Section 3.3, we stated that constraint solvers, which work only on string domain or only on non-string domain, are not effective for analyzing web applications. Thus, it is sufficient for us to compare S3 with only Kaluza and Z3-str.

3.6.1 Comparison with Kaluza

In this case study, we use the set of benchmarks that is shipped with Kaluza, which can be downloaded at:

`http://webblaze.cs.berkeley.edu/2010/kaluza`

They were generated using Kudzu [Saxena *et al.*, 2010], a symbolic execution framework for JavaScript, when testing 18 subject applications consisting of popular AJAX applications. The generated constraints are of boolean, integer and string types. Integer constraints also include ones on length of string variables, while string constraints include string equations, membership predicates.

Table 3.8 compares the performance and robustness of our solver S3 with Kaluza on the Kaluza benchmarks. Roughly speaking, this measures how fast and how often a solver is able to provide a *definitive answer*. This, in turn, means that if the solver returns SAT, then it should produce a particular *model* which demonstrates the executability of the path in question. If the solver returns UNSAT, then it should mean that the path in question is in fact not executable. There is of course a third possible case when the solver does not return any answer because of errors or non-termination, or returns UNKNOWN (when it neither find a model nor prove the unsatisfiability). A robust system therefore is one which returns definitive answers often.

	S3	Kaluza
Sat	34961	21651
Unsat	11799	23088
Error	0	2285
Timeout (20s)	524	340
Time (s)	16547	68768

Table 3.8: S3 versus Kaluza on Kaluza benchmarks

According to Table 3.8, S3 returns more SAT answers than Kaluza. Specifically, the difference is 13310 answers. For these benchmarks, Kaluza either returns UNSAT or gets errors. In addition, for the 524 benchmarks that S3 does not terminate, Kaluza also does not terminate or gets errors.

For each of the 34961 benchmarks, which S3 declares to be satisfiable, we conjoin the model generated by S3 with the original input formula and pass it to Kaluza. As a result,

Kaluza can now decide, with an answer confirming the satisfiability, even on those benchmarks that they could not decide before. In other words, all models produced by S3 are cross-checked by Kaluza.

We also use S3 to cross-check the models produced by Kaluza. Since in Kaluza, each query must specify a variable, for which they will generate the model, in our setting, we tested with the variable that starts with `var`[6]. As a result, Kaluza has errors with 11 benchmarks that do not have any variable starting with `var`. For the remaining 21640 benchmarks that Kaluza reports a SAT answer, the return model for 523 of them is incomplete. This is because given that Kaluza only returns the model for one variable, it is possible that the return model for the chosen variable may not be extensible to become a complete model which includes other variables. These 523 models are in fact not really models that are useful to reproduce attacks. (We note that [Zheng *et al.*, 2013] has previously remarked this "semi-soundness" issue of Kaluza.) This means that S3 has much more potential not only for vulnerability detection but also for attack reproduction than Kaluza does.

Lastly, Table 3.8 shows that S3 is more than 4 times faster than Kaluza. If we only take into account those benchmarks that Kaluza returns definitive answers, S3 is even more than 19 times faster than Kaluza. In short, S3 is far more efficient and robust than Kaluza.

3.6.2 Comparison with Z3-str

Recall that Z3-str deals with a *smaller* class of constraints than S3 (since Z3-str cannot handle regular expressions). The purpose of this study is to answer the question: w.r.t constraints that can be handled by *both* of the two solvers, are the performances the same? We now demonstrate that the answer is *no*, via defining the classes that show S3's improvement (esp. our enhanced design).

To demonstrate that S3 is better, we first use six test cases from the SAT benchmarks of Kaluza. We follow the setting of Z3-str as in [Zheng *et al.*, 2013] and remove all the constraints related to regular expressions. This way we can run Z3-str on the resulting constraints. These six benchmarks are presented in the first part of Table 3.9. For each of them, while S3 returns YES with a solution model, Z3-str instead returns NO. We note that the models S3 provides are validated as correct by using Z3-str itself.

We now briefly discuss why we have this difference. One reason is that Z3-str cannot

[6]There is usually one such variable in each benchmark.

Benchmark	Model produced?		Time(ms)		
	Z3-str	S3	Z3-str	S3	Z3-str/S3
ID_3482	NO	YES	_	58	_
ID_3468	NO	YES	_	23	_
ID_1543(*)	NO	YES	_	36	_
ID_3464	NO	YES	_	35	_
ID_3487	NO	YES	_	31	_
ID_new.23484(*)	NO	YES	_	21	_
sat_bnd	YES	YES	3225	120	27x
sat_unbnd	YES	YES	451s	129	3496x
unsat_bnd	_	NO	TO	30	_
unsat_unbnd	_	NO	TO	46	_

Timeout (TO) is at 2h. '*': regular expressions are removed.

Table 3.9: S3 versus Z3-str

acquire the concrete values assigned to length variables. In contrast, our design, presented in Section 3.4.2, enables the direct interactions between the string solver plug-in Z3-str-$\star$ and Z3 core, to query if the lengths of some string variables have been deduced or constrained in the arithmetic theory. This helps Z3-str-$\star$ avoid repetitive case analysis.

More specifically, the six we use in Table 3.9, have the following (frequent) pattern: there exists at least one variable that is only constrained by its length. Basically, with the constraint **length**(x)=i, the solution for x can be any string of length i, i.e. "@..@", where each @ is an arbitrary character. However, Z3-str cannot make use of this length constraint and keeps trying to assign string value for x, starting from the empty string. Given that x is constrained by its length, Z3-str must try-and-test many times until there is no more conflict with that length constraint. Thus, the total number of values to be tested by Z3-str will be blown up, preventing it from finding a solution.

We next consider another set of benchmarks, representing another pattern (which is also frequent in Kaluza's benchmarks): there exists a relationship between the lengths of different string variables. Indeed the example presented in Fig. 3.4 resembles such pattern. See the second part of Table 3.9, where statistics for 4 benchmarks are shown. We purposely make two benchmarks satisfiable – names start with 'sat', whereas the other two are unsatisfiable – names start with 'unsat'. In the two whose names end with 'bnd', the lengths of the string variables are bounded by 10, while in the other two (the names end with 'unbnd'), there is no such bound. For each satisfiable benchmarks, both Z3-str and S3 can find a correct solution model. However, S3 outperforms Z3-str significantly by an order of magnitude. For the unsatisfiable cases, while S3 returns NO within a second, Z3-str runs for more than 2

hours without producing an answer.

In summary, our design allows the full interaction between string theory and arithmetic theory, enabling S3 to handle length constraints more effectively. Thus, even discounting the fact that S3 solves a more general class of constraints than Z3-str, its performance is much better in the common class of constraints. This ensures its applicability in web programs, where length constraints are ubiquitous.

3.7 Related Work

Symbolic execution has recently been exploited to address a wide range of security problems. Some notable examples are: automated fingerprint generation [Brumley *et al.*, 2007], protocol replay [Newsome *et al.*, 2006], automated code transformation to eliminate SQL injection attacks in legacy web applications [Bisht *et al.*, 2010].

Motivated by the problem of analyzing JavaScript code for the purpose of detecting security flaws, [Saxena *et al.*, 2010] proposed a framework, Kudzu, which leverages the benefits of both concrete and symbolic evaluation. This work effectively reduced the analysis problem of web applications to the problem of solving string constraints. In order to be widely applicable, it is important to have a string solver which is able to reason about both string and non-string constraints. Importantly, the solver must also support constraints involving regular expressions and with multiple variables.

There is a vast literature on the problem of string solving. In previous Sections, we have carefully positioned our work against Kaluza and Z3-str. We now focus on other closely related work.

Practical methods for solving string equations can loosely be divided into bounded and unbounded methods. Bounded methods (e.g., HAMPI [Kiezun *et al.*, 2009a], CFGAnalyzer [Axelsson *et al.*, 2008], and [He *et al.*, 2013]) often assume fixed length string variables, then treat the problem as a normal constraint satisfaction problem (CSP). These methods can be quite efficient in finding satisfying assignments and often can express a wider range of constraints than the unbounded methods. However, as also identified in [Saxena *et al.*, 2010], there is still a big gap in order to apply them to constraints arising from the analysis of web applications.

In the spirit of Kaluza, [Bjørner *et al.*, 2009] proposed to reason about feasibility of

a symbolic execution path from high-level programs, of which string constraints are involved. In principle, the approach is similar to Kaluza: it proceeds by first enumerating concrete length values, before encoding strings into bit-vectors. It supports common integer related string operations, taken from the basic **.NET** string library, except for `replace`. Unlike Kaluza, however, regular expressions are not supported here. In a similar manner, [Redelinghuys *et al.*, 2012] addresses multiple types of constraints for Java PathFinder. Though this approach can handle many operators, it provides limited support for `replace`, requiring the result and arguments to be concrete. Furthermore, it does not handle regular expressions. In summary, the above methods are less powerful than S3 in terms of the expressiveness of the input language. Importantly, they have similar limitations as Kaluza, which we have carefully discussed.

PISA [Tateishi *et al.*, 2013] is the first path- and index-sensitive string solver that targets static analysis of web applications. The verification is conducted by encoding the program in Monadic Second-Order Logic (M2L). It supports regular expressions as well as Java's `replace` method. However, it does not support binary operations between two variables, i.e., PISA requires at least one of them to be constant. Also importantly, its expressiveness for arithmetic operations is restricted due to the limitations of M2L. For example, it does not support numeric multiplications and divisions.

Other unbounded methods are often built upon the theory of automata or regular languages. We will be brief and mention a few notable works. Java String Analyzer (JSA) [Christensen *et al.*, 2003] applies static analysis to model flow graphs of Java programs in order to capture dependencies among string variables. A finite automata is then derived to constrain possible string values. The work [Shannon *et al.*, 2009] used finite state machines (FSMs) for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the `java.lang.String` class, and some other related classes. They partially integrate a numeric constraint solver. For instance, string operations which return integers, such as `indexOf`, trigger case-splits over all possible return values.

In short, using automata and/or regular language representations potentially enables the reasoning of infinite strings and regular expressions. However, most of existing approaches have difficulties in handling string operations related to integers such as `length` and `indexOf`, let alone other high-level operations addressed in this work. More importantly, to assist web application analysis, it is necessary to reason about both string and

non-string behavior together. It is not clear how to adapt such techniques for the purpose, given that they do not provide native support for constraints of the type integer.

Since our method does not rely on the length bounds in enumerating solutions, and our particular treatment of (possibly unbounded) recursive operations is lazy, it is possible that S3 can handle query of unbounded length variables as well as unbounded regular expression. However, to guarantee termination, we do rely on the fact that the lengths are bounded. In fact, our work *targets* the input constraints arising from realistic web applications. Therefore, even when the lengths are not precisely known – in the case of static analysis – it is reasonable to assume that the lengths of input string variables are indeed bounded, as many modern practical string solvers do.

3.8 Concluding Remarks

This work presents a new algorithm for solving string constraints. The class of constraints is practically expressive, for its intended purpose of analyzing web programs which manipulate string inputs. Experimental evaluations show that our solver S3, despite being more expressive than other solvers, is much more robust and efficient.

We remark that in lieu of presenting an end-to-end system, we show that our proposed solver is indeed a *modular* contribution to any hypothetical dynamic symbolic execution end-to-end system. That is, the superior performance of our solver can be used, *without significant engineering* of integrating it, to obtain an improvement in the hypothetical system.

We believe, based on its symbolic representation of string constraints, S3 can also be extended to be more efficient in the context of static analysis, where even regular expressions can also be symbolically constructed.

Astute readers might already notice that our underlying symbolic representation goes well beyond regular languages. As an example, $\{a^n \cdot b^n \mid n \geq 0\}$ can be easily modeled as $\mathbf{star}(a, n) \cdot \mathbf{star}(b, n) \wedge n \geq 0$. While this work focuses on the practical impact of S3, investigating the theoretical impact of such symbolic representation is left as our future work.

Chapter 4

Progressive Reasoning

In this chapter, we continue the focus on solving string constraints, which is motivated by the security analysis of web applications. In the previous chapter, we have already introduced the lazy reasoning technique for string solving. However, since this technique does not address the non-termination issues, we now propose a novel progressive reasoning technique. Similarly to lazy reasoning, progressive reasoning is able to mitigate the problem of combinatorial search explosion. More importantly, it aims at a more complete search algorithm. In addition to presenting our algorithm, we also discuss the challenges of its implementation in the state-of-the-art SMT solver Z3. Finally, we demonstrate its applicability by testing the new string solver with a larger class of real-world benchmark programs.

4.1 Introduction

Web applications provide critical services over the Internet and handle sensitive data. Unfortunately, many of them are vulnerable to attacks by malicious users. According to the Open Web Application Security Project [OWASP, 2013], the most serious web application vulnerabilities include: (#1) Injection flaws (such as SQL injection) and (#3) Cross Site Scripting (XSS) flaws. Both vulnerabilities involve string-manipulating operations and occur due to inadequate sanitisation and inappropriate use of input strings provided by users. Therefore, reasoning about *strings* is necessary to ensure the security of web applications [Saxena *et al.*, 2010; Trinh *et al.*, 2014].

In web applications, recursively defined string functions also play an important role. For example, the string function **replaceAll** which is used frequently in sanitizers in order to prevent insecure user inputs, can be recursively defined as follows:

$$\begin{aligned}\texttt{Y=}\mathbf{replaceAll}\texttt{(X,r,Z)} \stackrel{def}{=}\ & \texttt{(X} \notin \texttt{/.}^{\star}\ \texttt{r}\ \texttt{.}^{\star}\texttt{/} \wedge \texttt{Y=X)} \vee \\ & \texttt{(X=}\mathtt{X}_1\texttt{·}\mathtt{X}_2\texttt{·}\mathtt{X}_3\texttt{·}\mathtt{X}_4 \wedge \mathtt{X}_2\texttt{·}\mathtt{X}_3 \in \texttt{/r/} \wedge \mathbf{length}\texttt{(}\mathtt{X}_3\texttt{)=1} \wedge \\ & \mathtt{X}_1\texttt{·}\mathtt{X}_2 \notin \texttt{/.}^{\star}\ \texttt{r}\ \texttt{.}^{\star}\texttt{/} \wedge \texttt{Y=}\mathtt{X}_1\texttt{·Z·}\mathtt{Y}_1 \wedge \mathtt{Y}_1\texttt{=}\mathbf{replaceAll}\texttt{(}\mathtt{X}_4\texttt{,r,Z))}\end{aligned}$$

The first disjunct corresponds to the base case where the input X does not contain any substring that matches the regular expression r. The resulting string Y will be the same as X. In the other disjunct, the first substring of X that matches r is $\mathtt{X}_2\texttt{·}\mathtt{X}_3$. So we replace this substring by Z and then make a recursive call for the remaining part $\mathtt{X}_4$. (The greedy version, using as many characters as possible in the match against r, can be defined and treated in a similar manner.)

Unfortunately, reasoning about unbounded strings defined recursively is in general an *undecidable* problem. As a concrete example, string functions such as **replaceAll** that are applied to any number of occurrences of a string (even limited to single-character strings) would make the satisfiability problem undecidable [Buchi and Senger, 1988; Bjørner *et al.*, 2009]. We must therefore be content with an *incomplete* solution.

Even so, we do not yet have an algorithm that is plausibly effective in practice. To generally handle recursive functions, a state-of-the-art technique [Trinh *et al.*, 2014] is "unfold-and-consume" which is to incrementally reduce recursive functions via splitting (and/or unfolding) process, until their subparts are bounded with constant strings/characters to be consumed. This technique has shown very promising results. However, because the main purpose of [Trinh *et al.*, 2014] is vulnerability detection, i.e., generating attack inputs for each satisfiable query, and that every query is invoked with a timeout limit, there was less emphasis on the detection of *unsatisfiable* queries. By contrast, in the setting of program verification, or in using verification technologies to speed up concolic testing [Jaffar *et al.*, 2013; Avgerinos *et al.*, 2014], the problem of determining unsatisfiability becomes paramount. In short, we can no longer depend on a timeout, and must seek a *terminating* algorithm as far as possible.

The main contribution of this work is an algorithm whose goal is to determine if a string formula is unsatisfiable, and if not, to be able to generate a solution for it. The key feature of our algorithm is a *pruning method* on the subproblems, in a way that is *directed*.

More specifically, our algorithm aims to detect non-progressive scenarios (Section 4.4.2) with respect to a criterion of minimizing the "lexicographical length" of the returned solution, if a solution in fact exists. Informally, in the search process based on reduction rules, we can soundly prune a subproblem when the answer we seek can be found more efficiently *elsewhere*. If a subproblem is deemed non-progressive, it means if the original input formula is satisfiable, then another satisfiable solution of shorter "length" will be found. If, on the other hand, the input formula is unsatisfiable, then any pruning is obviously sound. A technical challenge we will overcome is that at the point of pruning, the satisfiability of the input formula is *unknown*.

An additional important feature of our algorithm is applicable only when the input formula is unsatisfiable. Here, we want to produce a set of *conflict clauses*, a generalization of the input formula, that is now known to be unsatisfiable (Section 4.5.2). The benefits of such learning is of course well-known. It is, for example, at the heart of the attractiveness of SMT solvers. However, the key technical challenge is, how conflict clause learning can work in tandem with the pruning of non-progressive formulas, because at the time of pruning, again, the unsatisfiability of the input formula is unknown.

Finally, we present an experimental evaluation with two case studies. First is on the well-known Kudzu benchmark [Saxena *et al.*, 2010] where we show that (a) our new algorithm surpasses four state-of-the-art solvers in its ability to detect unsatisfiable formulas or generate a model in satisfiable formulas (and in good running time), and (b) the number of unsatisfiable cores is very small, thus paving the way to accelerate the consideration of large collections of formulas. The second case study considers web applications used in the Jalangi framework [Sen *et al.*, 2013], and shows how we can deal with the **replaceAll** operation in string formulas. No other system has been demonstrated on this class of problems, and thus the purpose of our evaluation is simply to show that we are applicable.

4.2 Motivation

The common reason for non-termination in string solving is non-progression. For example, after applying some reduction steps, if the reduced problem is not easier to solve than the original one, then it may lead to non-terminating computations. To illustrate, let us first look at the JavaScript example in Figure 4.1.

```
1  function json_decode(str) {
2    str = str.replace(/ip/g, "ip address");
3    str = str.replace(/dom/g, "domain");
4    return str;}
5  function json_show(str) {
6    var arr = JSON.parse(str);
7    var c = arr[0].content.split("&");
8    var s = c[0]+" "+c[1];
9    document.getElementById("info").innerHTML = s;}
10 res = json_decode(input);
11 json_show(res);
```

Figure 4.1: A JavaScript example using `replace` operation

The program takes as its input a JSON [ECMA-404,] string. Here is an example of a string input:

[{"content" : "ip=1.1.1.1&dom=nus.edu.sg" },
{"content" : "ip=0.0.0.0&dom=google.com" }]

Specifically, we store the JSON data in an array. Each element of the array is an object. Inside an object, we declare a property with its name and its value (i.e., a {name : value} pair). To access the value, we simply refer to the name of the property we need (e.g., we use `a`[0].content to access the value of the first element of the array `a`). In Figure 4.1, the program first decodes the input string by replacing all occurrences of `"ip"` with `"ip address"` and `"dom"` with `"domain"`. Then it parses the decoded string into an array `arr`, and splits the value of the first element of this array into two parts using "`&`" delimiter. Finally, it shows the resulting string `s` in a web browser by updating the `innerHTML` attribute of the `info` element.

Now, suppose we want to detect XSS vulnerabilities in the program. We then need to determine the security sink and source of XSS attacks. Here, the security sink is `innerHTML`, while the corresponding source is an input JSON string (i.e. `input`). Next, against the sink, we define the specification for XSS attacks which is some (regular) grammar encoding a set of strings that would constitute an XSS attack. For simplicity, we choose: all the strings that contain `"<script"`. Lastly, in order to generate a test input that leads to an XSS attack, we will need to solve the formula:

$$\textbf{contains}(s,\texttt{"<script"}) \wedge \texttt{tmp=}\textbf{replaceAll}(\texttt{input,"ip","ip address"}) \wedge \texttt{res=}\textbf{replaceAll}(\texttt{tmp,"dom","domain"}) \wedge \texttt{arr=parse(res)} \wedge \texttt{c=}\textbf{split}(\texttt{arr[0].content,"\&"}) \wedge \texttt{s=c[0]}\cdot\texttt{" "}\cdot\texttt{c[1]}$$

To make it easier for presentation, we simplify the formula into:

$$\texttt{res=}\mathbf{replaceAll}\texttt{(input,"ip","ip address")} \wedge \mathbf{contains}\texttt{(res,"<script")}$$

If we now perform some intuitive steps of "unfolding" the definition of **replaceAll**, we will reduce the simplified formula into two disjuncts. Since the first one is unsatisfiable due to the conflict between $\texttt{res} \notin \texttt{/.*\ "ip"\ .*/}$ and $\mathbf{contains}\texttt{(res,"<script")}$, we proceed to find a solution in the second disjunct, that is

$$\begin{gathered}\texttt{input=X}_1\texttt{·"ip"·input}_1 \wedge \texttt{X}_1\texttt{·"i"} \notin \texttt{/.*\ "ip"\ .*/} \wedge \\ \texttt{res=X}_1\texttt{·"ip address"·res}_1 \wedge \\ \texttt{res}_1\texttt{=}\mathbf{replaceAll}\texttt{(input}_1\texttt{,"ip","ip address")} \wedge \\ \mathbf{contains}\texttt{(res,"<script")}\end{gathered}$$

After applying the unfolding step some $n-1$ times, we still have to find a solution in the following formula:

$$\begin{gathered}\texttt{input=X}_1\texttt{·"ip"·input}_1 \wedge \texttt{X}_1\texttt{·"i"} \notin \texttt{/.*\ "ip"\ .*/} \wedge \\ \texttt{res=X}_1\texttt{·"ip address"·res}_1 \wedge \texttt{input}_1\texttt{=X}_2\texttt{·"ip"·input}_2 \wedge \\ \texttt{X}_2\texttt{·"i"} \notin \texttt{/.*\ "ip"\ .*/} \wedge \texttt{res}_1\texttt{=X}_2\texttt{·"ip address"·res}_2 \wedge \ldots \wedge \\ \texttt{res}_n\texttt{=}\mathbf{replaceAll}\texttt{(input}_n\texttt{,"ip","ip address")} \wedge \mathbf{contains}\texttt{(res,"<script")}\end{gathered}$$

Obviously, this will lead us to a non-terminating computation.

As a matter of fact, non-termination is common in string solving. In addition to the case of solving constraints on (JavaScript) recursive string operations (e.g. **replaceAll**, **split**, **match**), we also have non-termination when handling membership predicates with unbounded Kleene-star regular expressions.

Example 1. *Unbounded regular expressions:*

$$\begin{gathered}\texttt{X=Y·Z·T} \wedge \texttt{Y} \in \texttt{/a*/} \wedge \texttt{Z} \in \texttt{/b*/} \wedge \texttt{T} \in \texttt{/c*/} \wedge \\ \mathbf{length}\texttt{(Y)=}\mathbf{length}\texttt{(Z)} \wedge \mathbf{length}\texttt{(Z)=}\mathbf{length}\texttt{(T)} \wedge \texttt{X=X}_1\texttt{·"d"·X}_2\end{gathered}$$

Since the first 6 constraints state that `X` can be any string in the context-sensitive language $\{\ a^n{\cdot}b^n{\cdot}c^n \mid n{\geq}0\ \}$, automata techniques and the alike which approximate strings using context free grammars, are not able to handle this example. Instead, to generally deal with unboundedness of regular expressions which are constructed by using Kleene-star operators,

state-of-the-art techniques [Trinh *et al.*, 2014; Zheng *et al.*, 2015] represent the membership predicate `X∈/a*/` as an equation between string variable `X` and **star**(a,N) function which can be defined recursively as below:

$$\texttt{X=}\mathbf{star}\texttt{(a,N)} \stackrel{def}{=} \texttt{(X = "")} \vee \texttt{(X=a·}\mathbf{star}\texttt{(a,M)} \wedge \texttt{N=M+1)}$$

To facilitate the solving process, [Trinh *et al.*, 2014; Zheng *et al.*, 2015] will need to apply the definition of **star** functions to incrementally reduce them (according to the unfold-and-consume technique). However, they cannot handle Example 1 as they will go into an infinite loop of searching for a solution. We will discuss this example more in Section 4.4.

Finally, we note that the problem of non-terminating reasoning is not solely due to the recursive definitions we employ in this work. For example, the non-termination problem also happens when we do splitting on unbounded string variables. Below is a well-known example.

Example 2. *Overlapping variables:*

$$\texttt{X · "a" = "b" · X}$$

The classic work [Makanin, 1977] is able to solve the satisfiability problem of word equations (and not including recursively defined string operations). In this work, the big advance was to discover a termination criteria within the reasoning steps, and prominent amongst these was the "splitting" step. For the above example, such a step would split `X` in the left hand side to obtain a new formula `X·"a"="b"·X` ∧ `X="b"·Y` . This can then be simplified into `Y·"a"="b"·Y` ∧ `X="b"·Y` . Notice that the last formula is, in some sense, equally difficult to solve as the original one. The huge contribution of [Makanin, 1977] was thus to provide a bound for the number of times such "non-progressive" steps that needs to be made. However, the elaboration of this bound is extremely complex and is not considered feasible for a direct implementation.

4.3 The Core Language

We introduce the core constraint language in Figure 4.2. In our implementation, the string theory solver is a component of Z3 solver [De Moura and Bjørner, 2008b]. Though Z3 supports more primitive types, we only mention string type and integer type in Fig. 4.2.

$$
\begin{array}{lcll}
Fml & ::= & Literal \mid \neg\, Literal \mid Fml \wedge Fml & \\
Literal & ::= & A_s \mid A_l & \\
A_s & ::= & T_{str} = T_{str} & \\
A_l & ::= & T_{len} \leq m & (m \in C_{int}) \\
T_{str} & ::= & a & (a \in C_{str}) \\
 & \mid & X & (X \in V_{str}) \\
 & \mid & \mathbf{concat}(T_{str}, T_{str}) & \\
 & \mid & \mathbf{replaceAll}(T_{str}, T_{regexpr}, T_{str}) & \\
 & \mid & \mathbf{star}(T_{regexpr}, M) & (M \in V_{int}, M{\geq}0) \\
T_{regexpr} & ::= & a & (a \in C_{str}) \\
 & \mid & (T_{regexpr})^\star \mid T_{regexpr} \cdot T_{regexpr} & \\
 & \mid & T_{regexpr} + T_{regexpr} & \\
T_{len} & ::= & m & (m \in C_{int}) \\
 & \mid & M & (M \in V_{int}) \\
 & \mid & \mathbf{length}(T_{str}) \mid \Sigma_{i=1}^{n}(m_i * T_{len}) &
\end{array}
$$

Figure 4.2: The Syntax of Our Core Constraint Language

Variables: We deal with two types of variables: V_{str} consists of string variables (X, Y, Z, T, and possibly with subscripts); and V_{int} consists of integer variables (M, N, P, and possibly with subscripts).

Constants: Correspondingly, we have two types of constants: string and integer constants. Let C_{str} be a subset of $\Sigma^\star$ for some finite alphabet Σ. Elements of C_{str} are referred to as string constants or constant strings. They are denoted by a, b, and possibly with subscripts. Elements of C_{int} are integers and denoted by m, n, and possibly with subscripts.

Terms: Terms may be string terms or length terms. A string T_{str} term (denoted D, E, and possibly with subscripts) is either an element of V_{str}, an element of C_{str}, or a function on terms. More specifically, we classify those functions into two groups: recursive and non-recursive functions. An example of recursive function is **replaceAll**, while an example of non-recursive function is **concat**. The concatenation of string terms is denoted by **concat** or interchangeably by $\cdot$ operator. For simplicity, we do not discuss string operations such as **match**, **split**, **exec** which return an array of strings. We note, however, these operations are fully supported in our implementation.

A length term (T_{len}) is an element of V_{int}, an element of C_{int}, **length** function applied to a string term, a constant integer multiple of a length term, or their sum.

In addition, $T_{regexpr}$ represents regular expression terms. They are constructed from string constants by using operators such as concatenation ($\cdot$), union ($+$), and Kleene star ($\star$). However, regular expression terms are only used as parameters of functions such as

replaceAll and **star**.

Following [Trinh *et al.*, 2014], we use the **star** function in order to reduce a membership predicate involving Kleene star to a word equation. The **star** function takes two parameters as its input. The first parameter is a regular expression term while the second is a non-negative integer variable. For example, $X \in (r)^\star$ is modelled as $X = \mathbf{star}(r, N)$, where N is a *fresh* variable denoting the number of times that r is repeated.

Literals: They are either string equations (A_s) or length constraints (A_l).

Formulas: Formulas (denoted F, G, H, I, and possibly with subscripts) are defined inductively over literals by using operators such as conjunction ($\wedge$), and negation ($\neg$). Note that, each theory solver of Z3 considers only a conjunction of literals at a time. The disjunction will be handled by the Z3 core. We use $\texttt{Var}(F)$ to denote the set of all variables of F, including bound variables.

Define L to be the quantifier-free first-order two-sorted language over which the formulas described above are constructed. This logic can be considered as equality logic facilitated with recursive and non-recursive functions, along with length constraints.

As shown in [Trinh *et al.*, 2014], to sufficiently reason about web applications, string solvers need to support formulas of quantifier-free first-order logic over string equations, membership predicates, string operations and length constraints. Given a formula of that logic, similarly to other approaches such as [Trinh *et al.*, 2014; Zheng *et al.*, 2015], our top level algorithm will reduce membership predicates into string equations where Kleene star operations are represented as recursive **star** functions. After such reduction, the new formula can be represented in our core constraint language L in Figure 4.2.

4.4 Algorithm

In Section 4.4.1, we first present the background and limitation of existing methods. In Section 4.4.2, we then present the foundations of our progressive algorithm, along with the formal statements about its soundness and semi-completeness. Implementation details are discussed later in Section 4.5.

4.4.1 Preliminaries

This work builds on top of the string solver S3 [Trinh *et al.*, 2014]. Essentially, the S3 solver is a string theory plug-in built into the Z3 SMT solver [De Moura and Bjørner, 2008b], whose architecture is summarised as follows. Z3 core component consists of three modules: the congruence closure engine, a SAT solver-based DPLL layer, and several built-in theory solvers such as integer linear arithmetic, bit-vectors. The congruence closure engine can detect equivalent terms and then classify them into different equivalence classes which are shared among all theory solvers. Each theory solver can consult the Z3 core to detect equivalent terms if needed. In particular, the string theory solver has a bi-directional interaction with a built-in integer theory solver [Trinh *et al.*, 2014; Zheng *et al.*, 2015].

In the string theory solver, the search for a solution is driven by a set of *rules*.

Definition 4.4.1 (Derivation Rule)**.** *Each rule is of the general form*

$$\textit{(RULE-NAME)} \quad \frac{F}{\bigvee_{i=1}^{m} G_i}$$

where F, G_i are conjunctions of literals[1]*, $F \equiv \bigvee_{i=1}^{m} G_i$, and $\texttt{Var}(F) \subseteq \texttt{Var}(G_i)$.* □

An application of this rule transforms a formula at the top, F, into the formula at the bottom, which comprises a number (m) of *reducts* G_i.

Definition 4.4.2 (Derivation Tree)**.** *A* derivation tree *for a formula F is obtained by applying a derivation rule R to F. If the rule produces the single reduct* false*, then the tree comprises the single node labelled with F. Otherwise, let the reducts of R be G_i, $1 \leq i \leq m$. Then the tree comprises a root node labelled with F and there are m child nodes, labelled with G_i, $1 \leq i \leq m$.* □

The concepts of descendant and ancestor nodes are defined in the usual way.

A derivation tree rooted at formula F is built using some search strategy. The search strategy used by Z3 is a form of Depth First Search. This importantly means that the process can be nonterminating even though there is a finite path leading to a satisfying assignment to the variables in F. In navigating the construction of the derivation tree, we backtrack when we encounter a `false` formula. If all the leaf nodes of a subtree rooted at F are `false`, we can decide that the formula F is unsatisfiable.

[1] As per Figure 4.2.

On the other hand, when we encounter a formula for which no derivation rules can be applied, we can in fact terminate and decide that F is satisfiable. To ensure the soundness of this step, we employ a standard procedure of instantiating steps which enumerates and thus performs a *brute-force* method. This method looks for satisfying assignments for all the string variables in the root nodes of a dependency graph for string variables — a string variable in a root node does not depend on the values of any string variables. Consequently, when we terminate and declare satisfiability, it also means that every string variable has been successfully grounded. This brute-force method is part of Z3-str, S3, Z3-str2, and is also adopted by this work. We will henceforth assume this method tacitly, and not discuss it further.

Note that we control the branching order in navigating the derivation tree by dictating the order of the rules to be applied, as well as the order in which the reducts to be considered. In general, this order can affect significantly the overall performance of the algorithm. However, because of the way our progressive algorithm works, and in particular because of its pruning step (introduced later), the choice of order becomes much less important. For this reason, when we present our algorithm in detail below, we shall not impose any order on the application of derivation rules.

We next discuss the set of rules used by our solver. Then we will illustrate the application of rules and show an example of the derivation tree later in Example 3. The set of rules is described in two parts:

- one-reduct rules: in Fig 4.3 and Fig 4.4;
- multi-reduct rules: in Fig 4.5.

We first describe the one-reduct rules in Figure 4.3. These rules are to propagate length constraints, so that these constraints can be sent to integer theory solver. They are triggered by the encounter with a string constant, a string variable, a concatenation, and a string equation. In the figure, we use $\texttt{Var}(F)$, $\texttt{Constant}(F)$, $\texttt{Concat}(F)$, and $\texttt{Equality}(F)$ to denote the set of variables, constants, concatenations, and equations of F respectively. Note that we need to mark them in those corresponding sets so that these rules are applied once for each constant, variable, concatenation, and equation.

We comment here that in a practical implementation, it is useful to have some more rules, for example, to deal with membership predicates and string operations. But for a more focused presentation, we shall not discuss them further.

$$\text{(L-CST)}\quad \frac{F}{F \wedge \mathbf{length}(a) = |a|} \quad a \in \texttt{Constant}(F) \text{ and } |a| \text{ is the length of a string } a$$

$$\text{(L-VAR)}\quad \frac{F}{F \wedge \mathbf{length}(X) \geq 0} \quad X \in \texttt{Var}(F)$$

$$\text{(L-CAT)}\quad \frac{F}{F \wedge \mathbf{length}(D \cdot E) = \mathbf{length}(D) + \mathbf{length}(E)} \quad D \cdot E \in \texttt{Concat}(F)$$

$$\text{(L-EQL)}\quad \frac{F}{F \wedge \mathbf{length}(D) = \mathbf{length}(E)} \quad D = E \in \texttt{Equality}(F)$$

Figure 4.3: Length Constraint Propagation Rules

Next, consider Figure 4.4 which shows three basic simplification rules. First, the (CON) rule is to detect a contradiction in the string theory. Second, the (SUB) rule is to substitute all variables X in F with C, where C is either grounded or semi-grounded. A string is *grounded* if it is a constant string. It is called *semi-grounded* if it is either a **star** function, or a concatenation of which at least a component is either grounded or semi-grounded. For example, "a" is grounded, while "a" $\cdot\, Y_2$ is semi-grounded. Finally, the (SIM) rule is to eliminate matching constant strings on both sides of an equation. For each formula in the derivation tree, only one rule is applied at a time. For each application, only one literal is considered at a time. For example, in (SUB) rule, only $X = C$ is involved. The choice of which literal to be involved is decided by Z3.

$$\text{(CON)}\quad \frac{F \wedge D = E}{\texttt{false}} \quad D, E \text{ are string terms and } D \neq E$$

$$\text{(SUB)}\quad \frac{F \wedge X = C}{F[X/C] \wedge X = C} \quad X \in \texttt{Var}(F) \text{ and } C \text{ is (semi-)grounded}$$

$$\text{(SIM)}\quad \frac{F \wedge a \cdot D \cdot b = a \cdot E \cdot b}{F \wedge D = E} \quad D, E \text{ are string terms}$$

Figure 4.4: Simplification Rules for String Constraints

We comment here that in our implementation, we do employ other specialized rules. For example, because the string theory solver also receives the information of length constraints from the integer theory solver, we can craft a more specialized instance of the (CON) rule of Figure 4.4 where a variant side condition is that the lengths of D and E are different.

Further, our implementation accommodates string operations such as **substring**, **indexOf**, with new simplification rules. Again, for presentation purposes, we shall not discuss these detailed rules further.

Finally, we present the remainder of our rules: multi-reduct rules, which we call *splitting* rules. Before proceeding, note that in the rules in Fig 4.3 and Fig 4.4, no disjunction is introduced. The disjunctions are only introduced in the splitting rules, which we will present in two parts: the unfolding (UNF) rules, and the variable-splitting (SPL) rules.

$$\text{(SPL-1)}\quad \frac{F \wedge D \cdot a = b \cdot E}{\bigvee_{i=1}^{min(|a|,|b|)} (F \wedge D = b_0^{|b|-i} \wedge E = a_i^{|a|}) \vee (F \wedge \exists X_1 : D = b \cdot X_1 \wedge X_1 \cdot a = E)}$$

$$\text{(SPL-2)}\quad \frac{F \wedge D_1 \cdot E_1 = a \cdot D_2}{\bigvee_{i=0}^{|a|-1} (F \wedge D_1 = a_0^i \wedge E_1 = a_i^{|a|} \cdot D_2) \vee (F \wedge \exists X_1 : D_1 = a \cdot X_1 \wedge X_1 \cdot E_1 = D_2)}$$

$$\text{(SPL-3)}\quad \frac{F \wedge D_1 \cdot E_1 = D_2 \cdot b}{\bigvee_{i=|b|}^{1} (F \wedge E_1 = b_i^{|b|} \wedge D_1 = D_2 \cdot b_0^i) \vee (F \wedge \exists X_1 : E_1 = X_1 \cdot b \wedge D_1 \cdot X_1 = D_2)}$$

$$\text{(UNF-}\star\text{1)}\quad \frac{F \wedge D_1 \cdot D_2 = \mathbf{star}(a, N) \cdot E_2}{(F \wedge D_1 \cdot D_2 = E_2 \wedge N = 0) \vee (F \wedge \exists M : D_1 \cdot D_2 = a \cdot \mathbf{star}(a, M) \cdot E_2 \wedge N = M+1)}$$

$$\text{(UNF-}\star\text{2)}\quad \frac{F \wedge D_1 \cdot D_2 = E_1 \cdot \mathbf{star}(a, N)}{(F \wedge D_1 \cdot D_2 = E_1 \wedge N = 0) \vee (F \wedge \exists M : D_1 \cdot D_2 = E_1 \cdot \mathbf{star}(a, M) \cdot a \wedge N = M+1)}$$

Figure 4.5: Split rules and Unfold rules for **star** functions

An *unfolding rule* applies the definition of a recursive function, replacing the head with the body that typically contains a number of disjuncts (cf. the `replaceAll` function presented in Section 4.2). We describe such a rule using an unfolding rule *schema* (UNF) for a recursive function E as follows:

$$\text{(UNF)}\quad \frac{F \wedge D_1 \cdot D_2 = E \cdot D_3}{\bigvee (F \wedge D_1 \cdot D_2 = E_i \cdot D_3)} \quad E \text{ is defined as } \bigvee E_i$$

A *variable-splitting rule* is used to split a string variable into sub-variables. We shall describe such a rule using a variable-splitting rule schema (SPL) as follows:

$$\text{(SPL)}\quad \frac{F \wedge D_1 \cdot D_2 = E_1 \cdot E_2}{\begin{array}{c}(F \wedge D_1 = E_1 \wedge D_2 = E_2) \vee (F \wedge \exists Z : D_1 = E_1 \cdot Z \wedge Z \cdot D_2 = E_2 \wedge \mathbf{length}(Z) > 0) \\ \vee (F \wedge \exists T : E_1 = D_1 \cdot T \wedge D_2 = T \cdot E_2 \wedge \mathbf{length}(T) > 0)\end{array}}$$

The specific instances of (SPL) and (UNF) rules used in this work are listed in Figure 4.5. There are 3 split rules to deal with string equations and 2 unfold rules for **star** functions. The notation a_i^j denotes the substring of a from bound i to j.

We now discuss the relationship between the splitting rules and the issue of non-termination. Intuitively, the aim of the splitting rules is to reduce/break the current formula into "sub-formulas", where the complexity is reduced. A problem arises when the rule reduces the current formula into sub-formulas, where the complexity is actually *not* reduced. In other words, even though we have reduced the formula, we are in fact not any closer in finding a satisfying solution nor in finding a proof for unsatisfiability. This is the main reason for non-termination.

Let us now illustrate, in more detail, the issue of non-termination. We use Example 3, a simplified version of Example 1. Here, non-termination comes from dealing with recursive function **star** which is used to represent Kleene star regular expressions. We note that both Example 3 and Example 1 address the same non-progression problem in dealing with unbounded strings. Our purpose in choosing Example 3 to present is for simplicity.

Example 3. *Recursive function* **star***:*

$$X = \mathbf{star}(\text{“}a\text{”}, N) \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$$

Figure 4.6 summarizes the main steps of solving Example 3. (For simplicity, we ignore existential variables.) Similarly to solving Example 1, here we also need to unfold the definition of $\mathbf{star}(\text{“}a\text{”}, N)$ function and normalize the formula to DNF. An application of the unfold rule (UNF-$\star$1) would result in a disjunction of two reducts:

$$X{=}\text{“”} \wedge X{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z \quad \text{and}$$
$$X{=}\text{“}a\text{”}{\cdot}\mathbf{star}(\text{“}a\text{”}, M) \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z$$

The first reduct leads to a contradiction:

$$\text{(CON)}\ \dfrac{\text{(SUB)}\ \dfrac{X{=}\text{“”} \wedge X{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z}{X{=}\text{“”} \wedge \text{“”}{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z}}{\texttt{false}}$$

This contradiction appears in the tree depicted in Figure 4.6, but is hidden in the part of the tree that was abbreviated away for brevity.

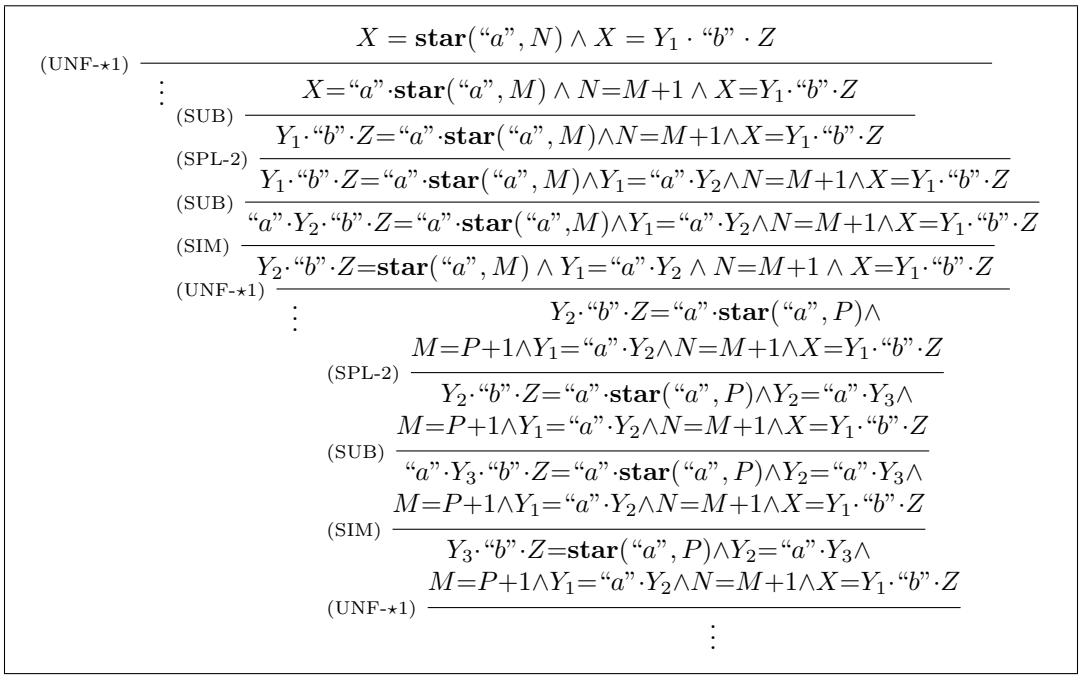


Figure 4.6: Derivation Tree for Example 3

In the second reduct, by substituting X with $Y_1 \cdot \text{“}b\text{”} \cdot Z$, we introduce a new constraint $Y_1 \cdot \text{“}b\text{”} \cdot Z = \text{“}a\text{”} \cdot \mathbf{star}(\text{“}a\text{”}, M)$. Now the only way to proceed is to split Y_1 into two parts: “a” and Y_2 (for brevity, we omitted the base case where $Y_1 = \text{“”}$). After substituting Y_1 with $\text{“}a\text{”} \cdot Y_2$ and simplifying the formula, we obtain a new constraint: $Y_2 \cdot \text{“}b\text{”} \cdot Z = \mathbf{star}(\text{“}a\text{”}, M)$. If we repeat this process of unfolding the definition of **star** function, clearly we will go into an infinite loop.

4.4.2 Progressive Search Strategy

As mentioned earlier, the key idea to achieve progression is to prune away a subtree when we are sure that a shorter solution can be found elsewhere. We first need to define a measure to decide which solution is shorter. This measure is parameterized by a sequence of variables. We use σ, τ to denote sequences.

Definition 4.4.3 (Lexical length of a solution)**.** *Given a formula F, let $\sigma = (x_1, x_2, \ldots, x_n)$ be a sequence of variables constructed from a non-empty subset of $\texttt{Var}(F)$. For each solution α of F, i.e. α is an assignment $[x_1 = a_1,\ x_2 = a_2, \ldots, x_n = a_n, \ldots]$, the lexical length of α is defined as a n-tuple $(\mathbf{length}(a_1), \mathbf{length}(a_2), \ldots, \mathbf{length}(a_n))$. We use $\texttt{Len}_\sigma(\alpha)$ to denote the lexical length of α w.r.t. the sequence σ.* □

We now use a lexical order to sort the solution set of a formula F based on the lexical length of each solution. If F has a solution then its minimum lexical length w.r.t. a sequence σ, denoted by $l(\sigma, F)$, is defined as the lexical length of a minimal solution of F. If F has no solution then its minimum lexical length is denoted by $\top$. We assume that $\forall \sigma, F : l(\sigma, F) \leq \top$. We now can compare two arbitrary formulas based on their minimum lexical length of solutions.

Definition 4.4.4 (Total order for formulas)**.** *Given two formulas F and G and let σ be a sequence of variables constructed from a non-empty subset of the common variables of F and G, a total order $\preceq_\sigma$ is defined as follows:*

$$F \preceq_\sigma G \stackrel{def}{=} l(\sigma, F) \leq l(\sigma, G) \qquad \square$$

We define equality $=_\sigma$ and strict inequality $\prec_\sigma$ in the obvious way. We now outline important properties of $\prec_\sigma$:

- [Prop-0]:
  If $F \Rightarrow G$ then for all sequence σ such that $\emptyset \subset \texttt{Var}(\sigma) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G)$, we have $G \preceq_\sigma F$

  Proof. Since $F \Rightarrow G$, any solution of F will be a solution of G. Therefore, for any sequence σ such that $\emptyset \subset \texttt{Var}(\sigma) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G)$, the minimal solution of F w.r.t. σ will be a solution of G. So for all such σ, $l(\sigma, G) \leq l(\sigma, F)$. By Definition 4.4.4, for all such σ, we have $G \preceq_\sigma F$. $\square$

- [Prop-1]:
  If $F \equiv (G \vee H)$ where $\texttt{Var}(F) \subseteq \texttt{Var}(H)$ and $\exists \sigma : F \prec_\sigma G$ then $F =_\sigma H$

  Proof. Since $\exists \sigma : F \prec_\sigma G$, then F must be satisfiable. Now let α be a minimal solution of F w.r.t. σ.

  With such α, and since $F \prec_\sigma G$, it follows that α is not a solution of G. Now, given $F \equiv (G \vee H)$, α must be a solution of H. This implies $F \not\prec_\sigma H$.

  Further, since $H \Rightarrow F$, by [Prop-0], for all τ such that $\emptyset \subset \texttt{Var}(\tau) \subseteq \texttt{Var}(F) \cap \texttt{Var}(H)$, we have $F \preceq_\tau H$. Since $\emptyset \subset \texttt{Var}(\sigma) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G) \subseteq \texttt{Var}(F) = \texttt{Var}(F) \cap \texttt{Var}(H)$, we have $F \preceq_\sigma H$.

  Finally, all the above culminates into $F =_\sigma H$. $\square$

- [Prop-2]:
  If $(G \vee H) \Rightarrow F$ and $\exists \sigma : F =_\sigma G$ then $F =_\sigma (G \vee H)$

  Proof. Let σ be such that $F =_\sigma G$. Given $(G \vee H) \Rightarrow F$, by [Prop-0], for all τ such that $\emptyset \subset \texttt{Var}(\tau) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G \vee H)$, we have $F \preceq_\tau (G \vee H)$. Since $\emptyset \subset \texttt{Var}(\sigma) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G) \subseteq \texttt{Var}(F) \cap \texttt{Var}(G \vee H)$, this in turn implies $F \preceq_\sigma (G \vee H)$.

  Also, $F =_\sigma G$ implies $F \not\prec_\sigma (G \vee H)$.

  Thus, $F =_\sigma (G \vee H)$. □

- [Prop-3]:
  If $\exists \sigma : F =_\sigma G$ and τ is a prefix of σ then $F =_\tau G$

  Proof. Let σ be such that $F =_\sigma G$. Suppose $l(\tau, F)$ is $(i_1, \ldots, i_{|\tau|})$. Then, because τ is a prefix of σ, we have $l(\sigma, F) = (i_1, \ldots, i_{|\tau|}, j_1, \ldots, j_{|\sigma|-|\tau|})$ for some $j_1, \ldots, j_{|\sigma|-|\tau|}$. Since $F =_\sigma G$, then $l(\sigma, G)$ is $(i_1, \ldots, i_{|\tau|}, j_1, \ldots, j_{|\sigma|-|\tau|})$. It follows that $l(\tau, G)$ is $(i_1, \ldots, i_{|\tau|})$. Therefore, $F =_\tau G$. □

Among them, we want to direct the attention towards the third property. It is used to ensure the soundness of the proposed method later. It states that if two formulas F and G have the same minimum lexical length of solutions w.r.t. a sequence σ, then they also have the same minimum lexical length of solutions w.r.t. a sequence τ, where τ is a prefix of σ.

Now we show how to prune a derivation subtree when we are sure that a solution with shorter lexical length can be found elsewhere. We do this by augmenting the strategy already described in Section 4.4.1 with a new step which enables us to prune the proof tree.

Definition 4.4.5 (Progressive Pruning)**.** *Let there be a derivation tree rooted at an input formula I, and let τ be a sequence of all the variables of I. Let F be a formula labelling a node in the tree. A set of prunable subtrees of F is a set of its descendants G_i such that there exists a sequence σ constructed from all variables of F satisfying the two conditions:*

- *τ is a prefix of σ and*
- *$F \prec_\sigma G_i$.*

We then prune *derivation subtrees rooted at formulas G_i.* □

The first condition ensures that a minimal solution of a formula F w.r.t. a sequence of all variables of F is also a minimal solution of F w.r.t. a sequence of all variables of the input formula I (according to [Prop-3]). Meanwhile, the second condition ensures that whenever we prune G_i, we still preserve a minimal solution of formula F w.r.t. a sequence of all of the variables of F.

Input: I : *Fml* , τ : a sequence on $\texttt{Var}(I)$
Output: SAT/UNSAT

⟨1⟩ **if** solve(I, τ, $\emptyset$) **return** SAT **else return** UNSAT

function solve(H : *Fml*, σ_{I}: a sequence, γ: a list of pairs of a formula and a sequence)
⟨2⟩ **if** ($H \equiv \texttt{false}$) **return** `false`
⟨3⟩ **if** (there is no rule to apply) **return** `true`
⟨4⟩ $\bigvee G_i \leftarrow$ applyRule(H) /* Apply a derivation rule */
⟨5⟩ Let Υ be the set of all the reducts G_i
⟨6⟩ **foreach** reduct $G \in \Upsilon$ **do** /* Choose G by following Z3 heuristics */
⟨7⟩ **if** (G contains a recursive term or a non-grounded concatenation)
⟨8⟩ **if** ($\exists(F, \sigma) \in \gamma$ *s.t.* $F \prec_\sigma G$) **return** `false` /* PRUNE !!! */
⟨9⟩ Let σ_{H} be a sequence on $\texttt{Var}(H)$ s.t. σ_{I} is a prefix of σ_{H} /* COND 1 */
⟨10⟩ $\gamma \leftarrow \gamma \cup \langle H, \sigma_{\text{H}} \rangle$
⟨11⟩ **endif**
⟨12⟩ **if** solve(G, σ_{I}, γ) **return** `true`
⟨13⟩ **if** (G contains a recursive term or a non-grounded concatenation)
⟨14⟩ $\gamma \leftarrow \gamma \setminus \{H, \sigma_{\text{H}}\}$
⟨15⟩ **endfor**
⟨16⟩ **return** `false`
end function

Algorithm S3P: Progressive Search

We now present our algorithm as Algorithm S3P. Line 2 corresponds to the case when we find a contradiction. In Line 6, we iterate over the set of sub-formulas; the ordering between them is not important. (In fact, in our implementation, we simply follow the heuristics of Z3.) Line 8 represents the key feature of our algorithm; it implements our pruning step (by returning false). Line 9 prepares for the pruning of a descendant of the current formula H (by ensuring that the first condition of Definition 4.4.5 is met).

Theorem 4.4.1 (Soundness)**.** *Given an input formula I, if Algorithm S3P*

- *returns SAT: then I is satisfiable;*
- *returns UNSAT: then I is unsatisfiable.*

Proof. We assume that:

- the standard search strategy represented by *not employing* the pruning step is sound, and
- we employ a sound and complete integer theory solver.

In other words, we only need to prove the soundness of the pruning step. More specifically, we need to prove the soundness of the return in line 8 of Algorithm S3P. In case I is unsatisfiable, the pruning is trivially sound; otherwise, we proceed by proving that a minimal solution of I, w.r.t. a sequence τ of all of its variables, is always preserved in the (remaining) tree after a subtree is pruned.

Let F_0 be a formula in the derivation tree, and σ be a sequence of all the variables of F_0. Let $F_i(1{\leq}i{\leq}n)$ be other descendants of I, ie. not including F_0, such that $I{\equiv}\bigvee_{i=0}^{n} F_i$. Let $G_j(0{\leq}j{\leq}m)$ be descendants of F_0 such that $F_0{\equiv}\bigvee_{j=0}^{m} G_j$. Finally, let $H_1 \equiv \bigvee_{i=1}^{n} F_i$, and $H_2 \equiv \bigvee_{j=1}^{m} G_j$.

By the design our algorithm, specifically line 9, we have that τ is a prefix of σ. Now we prove that

$$\text{if } F_0 \prec_\sigma G_0, \text{ then } I =_\tau (H_1 \vee H_2) \tag{1}$$

Since F_0 is a formula in the derivation tree, then $F_0 \Rightarrow I$. Since $\texttt{Var}(I) \subseteq \texttt{Var}(F)$, by [Prop-0], we have $I \preceq_\tau F_0$, which can be separated into two cases:

- Case $I \prec_\tau F_0$. By [Prop-1], we have $I =_\tau \bigvee_{i=1}^{n} F_i$, ie. $I =_\tau H_1$.
- Case $I =_\tau F_0$. As $F_0 \prec_\sigma G_0$, by [Prop-1], we have $F_0 =_\sigma \bigvee_{i=1}^{m} G_i$, ie. $F_0 =_\sigma H_2$. By [Prop-3], we have $F_0 =_\tau H_2$. By transitivity, $I =_\tau F_0 \wedge F_0 =_\tau H_2$ implies $I =_\tau H_2$.

Since $(H_1 \vee H_2) \Rightarrow I$, and by [Prop-2], we have that property (1) holds in these two cases. □

It can be seen that the first condition of the pruning step is very important. It is used in the second case of the above proof, in order to have the deduction from $F_0 =_\sigma H_2$ to $F_0 =_\tau H_2$. Suppose $\texttt{Var}(\tau) = \{x_1, .., x_n\}$. The condition guarantees that if a minimal solution of F_0 w.r.t. σ is $[x_1 = a_1, .., x_n = a_n, y_1 = b_1, .., y_m = b_m]$, then a minimal solution of F_0 w.r.t. τ is $[x_1 = a_1, .., x_n = a_n]$. Similarly, a minimal solution of H_2 w.r.t. τ is also $[x_1 = a_1, .., x_n = a_n]$. As such, the deduction is correct.

We now consider the completeness of Algorithm S3P. Before we can formalize this property, we need to discuss the condition check in line 8. This check determines the lexical order between two formulas, and is by no means a primitive operation. In fact, we do not know if the check is, in general, decidable. Our completeness result below nevertheless assumes that we have a decision procedure for this check. Later, in Section 4.5.1, we present an implementation which, though not a decision procedure, is sound and practical. We follow this up in Section 4.6 with an experimental evaluation.

Theorem 4.4.2 (Semi-Completeness)**.** *Suppose the given input formula I is satisfiable. Then Algorithm S3P will return SAT,* and *produce a minimal solution w.r.t. some sequence τ of all the variables of I.*

Proof. We first prove that for every formula F in the derivation tree, Algorithm S3P will terminate and apply a finite number of splitting rules for F. We assume it is clear that an inifinite number of applications of rules (CON), (SUB) and (SIM) can only occur with an infinite number of splitting rules.

If F is unsatisfiable then the progressive algorithm will definitely detect that $I \prec_\tau F$. As such, it will return `false` in line 8 of Algorithm S3P. So there is no application of splitting rules for F.

If F is satisfiable, we prove the following: for every instance of splitting rules, in each recursive case, the lower bound of at least one string variable of the input formula is increased. We refer to this as property (2).

We call variables of the input formula original variables. We will consider only the case of the (SPL) rule; others have a similar proof.

The following (SPL) rule is triggered when a string variable is involved, as opposed to a general string term.

$$\frac{G \wedge X_1 \cdot X_2 = Y_1 \cdot Y_2}{\begin{array}{l}(G \wedge X_1{=}Y_1 \wedge X_2{=}Y_2) \vee (G \wedge \exists Z : X_1 = Y_1{\cdot}Z \wedge Z{\cdot}X_2 = Y_2 \wedge \mathbf{length}(Z){>}0) \\ \vee (G \wedge \exists T : Y_1 = X_1{\cdot}T \wedge X_2 = T{\cdot}Y_2 \wedge \mathbf{length}(T) > 0)\end{array}}$$

Suppose X_1, X_2, Y_1, Y_2 are original variables in I. Because the first reduct formula above does not introduce any new concatenation operation, we can consider this formula the "base

case" while the other two are "recursive cases" where the recursive terms are concatenations. In the second and third reduct formulas, the lower bounds of X_1 and Y_1 (resp.) are increased. Each of these two cases introduces two new concatenations, that is $Y_1{\cdot}Z, Z{\cdot}X_2$, and $X_1{\cdot}T, T{\cdot}Y_2$. These 4 concatenations involve two new variables Z and T.

If the following applications of (SPL) rule are involved with either Y_1, X_2, X_1, or Y_2, then property (2) continues to hold. Otherwise, if the following applications of (SPL) rule are involved with either Z or T, then the increase of the lower bounds of those new variables will lead to the increase of the lower bounds of the corresponding original variables. As such property (2) holds for the (SPL) rule.

After a finite number of applications of splitting rules, suppose we have $F \equiv \bigvee H_i$. Because of property (2), for every reduct H_i that contains recursive term(s) (or non-grounded concatenations), there exists an original variable X whose lower bound is greater than n, where n is the length of X in $l(\tau, I)$. This means all of those reduct formulas have to be discharged in line 8 of Algorithm S3P. In other words, there is no application of splitting rules for H_i. So Algorithm S3P terminates.

We prove the second part of this theorem by contradiction. W.l.o.g. suppose Algorithm S3P finds a solution in a formula F in the derivation tree rooted at I. Suppose it is not a minimal solution of I w.r.t. a sequence τ of all of its variables. Because the progressive algorithm definitely detect that $I \prec_\tau F$, F has already been pruned in line 8 of Algorithm S3P. This is clearly a contradiction. □

4.5 Implementation

We first show how to implement the pruning step of our search algorithm. Then we present the conflict clause learning for string theory, especially in the setting of Z3.

4.5.1 The Pruning Step

To implement the pruning step of the Algorithm S3P, we have to keep track of the set γ which contains pairs of the current formula H and some sequence σ_{H} of all of the variables of H. When backtracking, such pair will be removed from γ correspondingly. Let τ be the sequence of all of the variables of the input formula I. The sequence σ_{H} is constructed by concatenating the sequence τ with additional variables from $\texttt{Var}(H)$. Specifically, $\sigma_{\text{H}} = \tau \ll \delta$ where

$\texttt{Var}(\delta) = \texttt{Var}(H) \setminus \texttt{Var}(\tau)$. For Example 3, after the first unfolding:

$$\tau \text{ is } (N, X, Y_1, Z) \text{ and } \gamma \text{ is } \{(X = \mathbf{star}(\text{“}a\text{”}, N) \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z,\ \tau)\}.$$

We now show how to implement the condition check in line 8 of Algorithm S3P. Suppose the current formula is G, if

- we find a pair (F, σ) in γ and a substitution θ such that $G\theta \Rightarrow F$, and
- the substitution θ is a progressive substitution (as defined in Definition 4.5.1 below) w.r.t. a sequence σ.

then the condition check is satisfied. Obviously, θ must not introduce new conflicts in $G\theta$, which prevents $G\theta$ from being `false` trivially.

Definition 4.5.1 (A progressive substitution)**.** *Let G be a formula, and σ be a sequence of subset variables of G. A substitution θ is progressive w.r.t. a sequence σ if for every solution α of G, there exists a solution β of $G\theta$ such that $\texttt{Len}_\sigma(\beta) < \texttt{Len}_\sigma(\alpha)$.* □

For Example 3, in the second unfolding, the current formula is

$$G \equiv Y_2 \cdot \text{“}b\text{”} \cdot Z = \mathbf{star}(\text{“}a\text{”}, M) \wedge Y_1 = \text{“}a\text{”} \cdot Y_2 \wedge N = M+1 \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$$

Obviously, there exists $F \equiv X = \mathbf{star}(\text{“}a\text{”}, N) \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$ and a substitution $\theta = [M/N, N/N{+}1, X/\text{“}a\text{”} \cdot X, Y_1/\text{“}a\text{”} \cdot Y_1, Y_2/Y_1, Z/Z]$, such that the implication check $G\theta \Rightarrow F$ succeeds. Furthermore, the substitution θ is progressive w.r.t. the sequence τ, that is (N, X, Y_1, Z). This is because if $\mathbf{length}(N) = k$ in a solution α (if any) of G, we have $\mathbf{length}(M) = k - 1$. Then, we have $\mathbf{length}(N) = k - 1$ in the corresponding solution α' of $G\theta$. Because $\texttt{Len}_\tau$ function returns a 4-tuple whose first element is $\mathbf{length}(N)$, θ is progressive. As a result, we can stop the second unfolding.

Lemma 4.5.1. *The implementation of the pruning step is sound*

Proof. Let G be the current formula and F, σ, θ be such that the condition check is satisfied. According to the construction of σ, the sequence of additional variables of a formula F follows the variables of the input formula. Thus, the first condition of the pruning step is satisfied. For the second condition, we already know that G is a descendant of F. We now prove that $F \prec_\sigma G$.

By Definition 4.5.1, because θ is progressive, there exists a solution β of $G\theta$ such that $\texttt{Len}_\sigma(\beta) < l(\sigma, G)$. Next, because the implication check $G\theta \Rightarrow F$ succeeds, β is also a solution of F, which means $l(\sigma, F) \leq \texttt{Len}_\sigma(\beta)$. By transitivity, we have $l(\sigma, F) < l(\sigma, G)$. In short, we have $F \prec_\sigma G$. □

4.5.2 Conflict Clause Learning

We present our conflict clause learning technique for string theory, with the focus on the case when non-progression is detected. Specifically, in the implementation of the pruning step, suppose there exists (F, σ) in γ and a substitution θ such that $G\theta \Rightarrow F$ and θ is progressive w.r.t. σ. A corollary of Lemma 4.5.1 is that we have $F \prec_\sigma G$ (see the proof of Lemma 4.5.1). Now, in addition to returning `false` as in line 8 of Algorithm S3P, we also mark $\widehat{G}$ as a possible conflict clause. We derive $\widehat{G}$ from G by removing all equations in solved form which is defined for both string and integer theories as below. If later we can not find any solution in solving F, then we can conclude F is unsatisfiable and produce a conflict clause $\widehat{G}$. The soundness of this learning is stated in Lemma 4.5.2 and Lemma 4.5.3.

Definition 4.5.2 (String Solved Form)**.** *A string equation is in solved form if it is in the form of* $X{=}f(Y_1, ..., Y_n, a_1, ..., a_m)$*, where* $X \in V_{str}$*,* $Y_1, ..., Y_n \in V_{str}$*,* $a_1, ..., a_n \in C_{str}$*,* $X \notin \{Y_1, ..., Y_n\}$*, and* f *is a non-recursive function.* □

For example, $X{=}\mathbf{concat}(Y, Z)$ is in solved form. $X{=}\mathbf{concat}(Y, \mathbf{concat}(Y_1, Y_2))$ can be rewritten into two formulas $X{=}\mathbf{concat}(Y, Z)$ and $Z{=}\mathbf{concat}(Y_1, Y_2)$, which are both in solved form. Similarly, we can define a solved form in integer theory:

Definition 4.5.3 (Integer Solved Form)**.** *An equation is in solved form if it is in the form of* $M{=}g(N_1, .., N_n, p_1, ..., p_m)$*, where* $M{\in}V_{int}$*,* $V_1, .., V_n{\in}V_{int} \cup V_{str}$*,* $p_1, ..., p_m{\in}C_{int} \cup C_{str}$*,* $M \notin \{N_1, ..., N_n\}$*, and* g *is a function.* □

Now, suppose some formula G contains an equation $X{=}f(\cdots)$ in solved form, we are able to eliminate variable X by substituting X with $f(\cdots)$ in G. To obtain $\widehat{G}$, we need to remove all equations in solved form from G. The purpose of deriving $\widehat{G}$ is to obtain the core reason for pruning G, which helps us to extract a smaller unsatisfiable core for the input formula. For Example 3, G is $Y_2{\cdot}\text{“}b\text{”}{\cdot}Z = \mathbf{star}(\text{“}a\text{”}, M) \wedge Y_1{=}\text{“}a\text{”}{\cdot}Y_2 \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z$. So we have 3 equations $Y_1{=}\text{“}a\text{”}{\cdot}Y_2$, $N{=}M{+}1$, and $X{=}Y_1{\cdot}\text{“}b\text{”}{\cdot}Z$ which are in solved form. Therefore, we mark $\widehat{G} \equiv Y_2 \cdot \text{“}b\text{”} \cdot Z = \mathbf{star}(\text{“}a\text{”}, M)$ as a possible conflict clause. Later,

when we can decide the unsatisfiability of the input formula, based on the implication graph, we can trace back to extract an unsat core for the input formula. Specifically, it is $X = \mathbf{star}(\text{“}a\text{”}, N) \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$.

Lemma 4.5.2. *Suppose the pruning condition check is applied for specific formulas F and G. Then F can be written into the form $G \vee G_r$ and the following holds: if G_r is unsatisfiable, F is unsatisfiable.*

Proof. Similarly to the proof of Lemma 4.5.1, we can prove that $F \prec_\sigma G$. Also, by [Prop-1], we have $F =_\sigma G_r$. Therefore, if G_r is unsatisfiable, F is unsatisfiable. □

Lemma 4.5.3. *$\widehat{G}$ is satisfiable iff G is satisfiable.*

Proof. This lemma holds by construction (of $\widehat{G}$). □

Now we present the detailed implementation of obtaining $\widehat{G}$ in Z3, given that Z3 manages theory terms via its congruence closure engine. First, we give an overview on how Z3 builds its equivalence classes. Given an equation, its two sides will be represented as two nodes in an equivalence class. For Example 3, since G is $Y_2 \cdot \text{“}b\text{”} \cdot Z = \mathbf{star}(\text{“}a\text{”}, M) \wedge Y_1 = \text{“}a\text{”} \cdot Y_2 \wedge N = M+1 \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$, we have 4 equivalence classes as follows:

- $X \quad , \quad Y_1 \cdot \text{“}b\text{”} \cdot Z$
- $Y_2 \cdot \text{“}b\text{”} \cdot Z \quad , \quad \mathbf{star}(\text{“}a\text{”}, M)$
- $Y_1 \quad , \quad \text{“}a\text{”} \cdot Y_2$
- $N \quad , \quad M + 1$

Note that given a node e representing a term Q, we are able to access all nodes representing terms that take term Q as their parameters (e.g., for string term D and E, we can access the nodes representing $\mathbf{length}(D)$, $\mathbf{concat}(D, E)$). We call the later parent nodes of e.

There are three steps to remove an equation $V = f(\cdots)$ in solved form. First, we mark the node representing variable V. A node e is marked when:

- it represents a single variable V (V can be either a string variable or an integer variable),
- the size of its equivalence class is greater than 1,

- its parent nodes are not in the same equivalence class as e, and
- not all of remaining nodes in the equivalence class of e contain recursive functions.

Second, we substitute the value of all marked nodes in their parent nodes with the value of another node in the equivalence classes of the marked nodes. Finally, we need to traverse all unmarked nodes in the equivalence classes to create a conjunction of all equations. For Example 3, according to above conditions, nodes representing X, Y_1, and N will be marked in their corresponding equivalence classes. Then, we can traverse all unmarked nodes to obtain the formula $\widehat{G} \equiv Y_2 \cdot \text{“}b\text{”} \cdot Z = \textbf{star}(\text{“}a\text{”}, M)$.

4.6 Evaluation

We implemented our algorithm into S3 [Trinh *et al.*, 2014] which itself was built on top of the Z3 framework [De Moura and Bjørner, 2008b]. Our solver is called S3P which stands for *Progressive S3*. To evaluate our solver, we conduct two case studies which involve practical benchmark constraints generated from testing JavaScript web applications. All experiments are run on a 3.2GHz machine with 8GB memory.

In the first case study, we used a large and popular set of benchmark constraints generated using the Kudzu symbolic execution framework [Saxena *et al.*, 2010]. State-of-the-art string solvers are also evaluated using this benchmark suite, making it convenient for us to provide detailed comparisons on the applicability and efficiency of our new solver.

Note that the constraints in Kudzu's benchmarks have already been preprocessed and/or over-simplified. In particular, the string lengths have been bounded and recursive string function such as **replaceAll** have been transformed to primitive operators so that the underlying solver of Kudzu [Saxena *et al.*, 2010] can handle. Because strong support for the **replaceAll** function is critical for enhancing security analysis of web applications, we conduct a second case study, of a smaller scale, but with special focus on the **replaceAll** function. The main purpose is to show that S3P is more applicable than existing solvers in such domain applications.

Kudzu Benchmarks: In this case study, we use the set of constraints which can be downloaded at: http://webblaze.cs.berkeley.edu/2010/kaluza. They were generated using Kudzu [Saxena *et al.*, 2010], a symbolic execution framework for JavaScript, when testing

18 subject applications consisting of popular AJAX applications. The generated constraints are of boolean, integer and string types. Integer constraints also include ones on length of string variables, while string constraints include string equations, membership predicates. To compare with other solvers, we choose to use the SMT-format version of Kaluza benchmark as provided in [Liang *et al.*, 2014].

This case study consists of two parts. The first part is to evaluate our non-progression detection technique. Table 4.1 shows the result of solving Kudzu constraints by S3P, compared with 4 state-of-the-art solvers: Norn (v1.0), CVC4 (v1.4), S3 (v17092015), Z3-str2 (v1.0.0). While Norn is automata-based string solver, the others, including S3P, are word-based string solvers, in which string is treated as a basic type.

Table 4.1: Constraints generated by Kudzu

	Norn	CVC4	S3	Z3-str2	S3P
Sat	27068	33227	34961	34931	35270
Unsat	11561	11625	11799	11799	12014
Unk	0	0	0	524	0
Error	6187	0	0	0	0
TO (20s)	2468	2432	524	30	0
Time (s)	178960	50346	16547	6309	6972

It can be seen that automata-based solvers such as Norn are not good at handling constraints generated from concolic testing of web applications. This is because such constraints are usually of multi-sorted theory, including both string constraints and integer constraints, such as those coming from the string lengths.

In fact, for the case of Kudzu constraints, all word-based string solvers dominate Norn. Not counting S3P, Z3-str2 is the solver that produces the best result. Z3-str2 also terminates on 524 benchmarks where Norn, CVC4 and S3 all time out. Specifically, Z3-str2 terminates with an `Unknown` answer if the input formula contains the so-called "overlapping variables" [Zheng *et al.*, 2015].

Compared with Z3-str2, S3P can in fact decide the satisfiability of these 524 benchmarks. S3P achieves this by employing the proposed technique for non-progression detection. Specifically,

- if an input formula is unsatisfiable, S3P is able to decide the unsatisfiability of that formula. For example, it can decide the unsatisfiability of 215 input formulas in those 524 benchmarks.

- otherwise, being able to effectively prune away non-progressive paths, S3P has a chance of finding solutions in other search branches. As such, the remaining of those 524 benchmarks are decided as satisfiable with the correct models.

In fact, for each of the 35270 benchmarks which S3P declares to be satisfiable, we conjoin the model generated by S3P with the original input formula and pass it to the other 4 solvers. As a result, all 4 solvers can now decide, with an answer confirming the satisfiability, even on those benchmarks they could not decide before. In other words, all models produced by S3P are cross-checked and all the solvers reach a consensus for every single case.

Table 4.2: Usefulness of unsatisfiable cores for Kudzu framework

# unsat files		12014
S3P	Time	1129s
S3P with unsat core	# unsat cores	59
	% skipped	99.5
	Time	11s

In the second part of this case study, we focus on benchmarks which are unsatisfiable, in order to demonstrate our conflict clause learning technique. More specifically, we will extract the unsatisfiable cores from those input constraints, and show the potential usefulness of the cores in a dynamic symbolic execution (DSE) framework (e.g. Kudzu). To do this, we compare the result of solving 12014 unsatisfiable formulas in Kudzu benchmarks by two versions of S3P. The first version (S3P) will solve each formula independently. In contrast, when deciding a formula as unsatisfiable, the second version will cache its unsat core. Subsequently, it will attempt to skip a formula if the formula is discharged by some cached unsat core. The result is summarized in Table 4.2. There are two important observations:

- By extracting and caching the unsatisfiable cores of 59 formulas, we can skip checking the satisfiability of the remaining formulas (99.5%) (which in fact represent infeasible paths to the attack against the sink). Overall, we achieve the speedup of about 102x faster.

- Unsatisfiable cores are also useful for validating/debugging the result. By inspecting a much smaller number of constraints compared to the original ones, we are able to validate the final result. For example, we are able to confirm that all unsatisfiable answers are correct by inspecting them manually.

Jalangi Benchmarks: This second case study is to focus on the **replaceAll** string function. As such, we collect constraints generated by testing web applications using the concolic tester in Jalangi framework [Sen *et al.*, 2013], and do not make any preprocessing with those constraints. These applications are `annex`, `tenframe`, `calculator`, `go`, and `shopping`. Note that all of them are not vulnerable to XSS attacks.

Let us first present the set-up to collect this set of constraint benchmarks. For each web application, we choose a sink point, that is innerHTML. Then we symbolically execute paths from a source to the sink. These path constraints will be combined with attack specifications at the sink. The resulting formulas are sent to a constraint solver.

Table 4.3: Constraints generated by Jalangi

# benchmarks	# constraints	# **replaceAll** operation	Time of S3P
48	624	96	143.7 s

Table 4.3 summarizes the statistics of those formulas, along with the running time of S3P. In 48 benchmarks, there are 624 constraints and 96 constraints are involved in **replaceAll** operation. So the percentage of **replaceAll** operation is about 15%.

More importantly, **replaceAll** operation appears in all benchmarks. The reason is that after a source point, a web application usually provides some sanitizing mechanism, for example, by replacing all "<" with "&`lt;`" and ">" with "&`gt;`". As such, the path constraints usually involve the **replaceAll** function. For a concrete example, after symbolically executing the program, a DSE framework will combine the path constraints with the specifications for attacks, to create queries for the constraint solver. A specification for innerHTML sink can be all the strings that contain "$<$ `script`". Then a simplified example of a common pattern is:

$$\texttt{input}_1 = \textbf{replaceAll}(\texttt{input}, \text{“<”}, \text{“\<”}) \wedge \texttt{input}_2 = \textbf{replaceAll}(\texttt{input}_1, \text{“>”}, \text{“\>”}) \wedge$$
$$\texttt{output} = \texttt{input}_2 \cdot \text{“</br>”} \wedge \textbf{contains}(\texttt{output}, \text{“<script”})$$

Given that Z3-str2, CVC4, and Norn *cannot* deal with **replaceAll** operation, the only work which is comparable in term of the expressiveness as our solver, is S3. However, S3 *timeouts* for all of those formulas because it goes into infinite loops (similarly to what we have shown in Section 4.2). In contrast, S3P can decide the unsatisfiability of all benchmarks. Since S3P is the only solver that is applicable in those constraints (which are generated from testing web applications), we believe it will make a remarkable contribution to ensuring the security of web applications.

4.7 Related Work

There is a vast literature on the problem of string solving. Practical methods for solving string equations can loosely be divided into bounded and unbounded methods. Bounded methods (e.g., HAMPI [Kiezun *et al.*, 2009a], CFGAnalyzer [Axelsson *et al.*, 2008], and [He *et al.*, 2013]) often assume fixed length string variables, then treat the problem as a normal constraint satisfaction problem (CSP). These methods can be quite efficient in finding satisfying assignments and often can express a wider range of constraints than the unbounded methods. However, as also identified in [Saxena *et al.*, 2010], there is still a big gap in order to apply them to constraints arising from the analysis of web applications.

To reason about feasibility of a symbolic execution path from high-level programs, of which string constraints are involved, one approach [Saxena *et al.*, 2010; Bjørner *et al.*, 2009] is to proceed by first enumerating concrete length values, before encoding strings into bit-vectors. In a similar manner, [Redelinghuys *et al.*, 2012] addresses multiple types of constraints for Java PathFinder. Though this approach can handle many operators, it provides limited support for **replace**, requiring the result and arguments to be concrete. Furthermore, it does not handle regular expressions. In summary, all of them have similar limitations such as performance [Trinh *et al.*, 2014].

Unbounded methods are often built upon the theory of automata or regular languages. We will be brief and mention a few notable works. Java String Analyzer (JSA) [Christensen *et al.*, 2003] applies static analysis to model flow graphs of Java programs in order to capture dependencies among string variables. A finite automata is then derived to constrain possible string values. The work [Shannon *et al.*, 2009] used finite state machines (FSMs) for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the `java.lang.String` class, and some other related classes. They partially integrate a numeric constraint solver. For instance, string operations which return integers, such as **indexOf**, trigger case-splits over all possible return values. A recent work [Aydin *et al.*, 2015] provides an automata-based technique for solving string constraints and a method for counting the number of solutions to such constraints. In addition, string solver Norn [Abdulla *et al.*, 2014; Abdulla *et al.*, 2015] is also based on automata techniques. They have limited or no support for **replace** operations.

Using automata and/or regular language representations potentially enables the reasoning of infinite strings and regular expressions. However, most of existing approaches have

difficulties in handling string operations related to integers such as **length**, let alone other high-level operations addressed in this work. More importantly, to assist web application analysis, it is necessary to reason about both string and non-string behavior together. It is not clear how to adapt such techniques for the purpose, given that they do not provide native support for constraints of the type integer.

Most of recent works on string solving are based on unbounded methods with string as a primitive data type. Examples are Z3-str [Zheng *et al.*, 2013], CVC4 [Liang *et al.*, 2014; Liang *et al.*, 2015; Liang *et al.*, 2016; Barrett *et al.*, 2016], S3 [Trinh *et al.*, 2014], Z3-str2 [Zheng *et al.*, 2015]. However, none of them addresses the non-termination issues in string solving as in this work. Though in [Zheng *et al.*, 2015], the authors address non-termination in splitting overlapping string variables, they currently can not decide the satisfiability of such formulas. In contrast, we generalize common non-termination issues that appear in solving string constraints generated from reasoning about web applications. Along with that is a progressive algorithm which we believe is applicable to not just S3, but also other solvers in this family of word-based string solvers.

4.8 Conclusion

This work presents a progressive algorithm for solving string constraints for the intended purpose of analyzing practical web applications. Its main feature is its ability to handle the termination problem when unfolding recursive definitions which define the constraints. This, together with another feature of conflict clause learning, were demonstrated to show usefulness in pruning the search space and new levels of results in Javascript benchmarks arising from web applications. Finally, because our algorithm deals with recursive definitions in a somewhat general manner, we believe it can be extended to support reasoning about unbounded data structures, for example heap-allocated data structures.

Chapter 5

Inductive Reasoning

In Chapter 2 and Chapter 3, we have discussed lazy reasoning techniques for entailment proving and string solving respectively. The limitation of these techniques is that they do not address the non-termination issues. This motivates the introduction of progressive reasoning for string solving in Chapter 4, and inductive reasoning for entailment proving in this chapter. Specifically, we now propose a general proof method for recursive predicates that includes reasoning by induction. Our method helps us to automate the verification of a large class of heap-manipulating programs. We have evaluated our prototype implementation on a comprehensive set of benchmarks, including both academic algorithms and real programs.

5.1 Introduction

We consider the automated verification of imperative programs with emphasis on reasoning about the functional correctness of dynamically manipulated data structures. The dynamically modified heap poses a big challenge for logical methods. This is because typical correctness properties often require combinations of structure, data, and *separation*.

Automated proofs of data structure properties — usually formalized using Separation Logic (or the alike) and extended with *user-defined* recursive predicates — "rely on decidable sub-classes together with the corresponding proof systems based on (un)folding strategies for recursive definitions" [Navarro and Rybalchenko, 2011]. Informally, in the regard of handling recursive predicates, the state-of-the-art [Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Piskac *et al.*, 2013; Pek *et al.*, 2014], to name a few, collectively called unfold-and-match (U+M) paradigm, employ the basic but systematic transformation steps of *folding* and *unfolding* the rules.

A proof, using U+M, succeeds when we find successive applications of these transformation steps that produce a final formula which is *obviously* provable. This usually means that either (1) there is no recursive predicate in the RHS of the proof obligation and a direct proof can be achieved by consulting a generic SMT solver; or (2) no special consideration is needed on any occurrence of a predicate appearing in the final formula. For example, if $\mathtt{p}(\tilde{u}) \ \wedge \cdots \models \mathtt{p}(\tilde{v})$ is the formula, then this is obviously provable if $\tilde{u}$ and $\tilde{v}$ were *unifiable* (under an appropriate theory governing the meaning of the expressions $\tilde{u}$ and $\tilde{v}$). In other words, we have performed "formula abstraction" [Madhusudan *et al.*, 2012] by treating the recursively defined term $\mathtt{p}()$ as *uninterpreted*.

A key feature that is missing from the U+M methodology is the ability to prove by *induction*, which is often required in verification of practical examples [Berdine *et al.*, 2005]. Without inductive reasoning, U+M (folding/unfolding together with formula abstraction) *cannot* handle proof obligations involving *unmatchable* predicates. Specifically, in such obligations, there exists a recursively defined predicate in the RHS which cannot be transformed, via folding/unfolding, to one that is unifiable with some predicate in the LHS.

As a concrete example, consider the following definitions of list and list of zero numbers:

$$\begin{array}{rcl} \mathsf{vlist}(\mathtt{x}) & \stackrel{def}{=} & \mathtt{x{=}null} \ \wedge \ \mathbf{emp} \\ & & \mid (\mathtt{x}{\mapsto}\_, \mathtt{t}) \ * \ \mathsf{vlist}(\mathtt{t}) \\ \mathsf{zero\_list}(\mathtt{x}) & \stackrel{def}{=} & \mathtt{x{=}null} \ \wedge \ \mathbf{emp} \\ & & \mid (\mathtt{x}{\mapsto}\mathtt{0}, \mathtt{t}) \ * \ \mathsf{zero\_list}(\mathtt{t}) \end{array}$$

In Fig. 5.1, we present a partial proof that a list of zero elements is a list. First, by unfolding the LHS, the original proof obligation is resolved into (i) and (ii). The first sub-obligation can be easily discharged by unfolding the RHS. (It is clear that U+M is inadequate for this proof. This is because no matter how we apply folding/unfolding, there still exists a predicate vlist in the RHS, which cannot be matched with the predicate zero_list in the LHS.)

$$\dfrac{\dfrac{\dfrac{}{\mathtt{True}}\,(\textsc{obvious})}{\mathtt{x{=}null} \wedge \mathbf{emp} \models \mathtt{x{=}null} \wedge \mathbf{emp}}\,(\textsc{right-unfold})}{\mathtt{x{=}null} \wedge \mathbf{emp} \models \mathsf{vlist}(\mathtt{x})\ \text{(i)} \qquad (\mathtt{x}{\mapsto}\mathtt{0}, \mathtt{t}) * \mathsf{zero\_list}(\mathtt{t}) \models \mathsf{vlist}(\mathtt{x})\ \text{(ii)}}\,(\textsc{left-unfold})$$
$$\mathsf{zero\_list}(\mathtt{x}) \models \mathsf{vlist}(\mathtt{x})$$

Figure 5.1: Partial Proof Tree for $\mathsf{zero\_list}(\mathtt{x}) \models \mathsf{vlist}(\mathtt{x})$

Now let us consider the original proof obligation $\mathsf{zero\_list}(\mathtt{x}) \models \mathsf{vlist}(\mathtt{x})$ as an *induction hypothesis*. This justifies an *induction step* comprising a transformation of (ii) into a simpler

$$\dfrac{\dfrac{\dfrac{\texttt{True}}{\texttt{x=null} \wedge \textbf{emp} \models \texttt{x=null} \wedge \textbf{emp}}\,(\textsc{obvious})}{\texttt{x=null} \wedge \textbf{emp} \models \mathsf{zero\_list}(\texttt{x})\ (1)}\,(\textsc{right-unfold}) \qquad (\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models \mathsf{zero\_list}(\texttt{x})\ (2)}{\mathsf{vlist}(\texttt{x}) \models \mathsf{zero\_list}(\texttt{x})}\,(\textsc{left-unfold})$$

Figure 5.2: Partial Proof Tree for $\mathsf{vlist}(\texttt{x}) \models \mathsf{zero\_list}(\texttt{x})$

obligation, as follows: weaken the LHS by replacing $\mathsf{zero\_list}(\texttt{t})$ with $\mathsf{vlist}(\texttt{t})$, and obtain the new proof obligation (iii). It is now easy to prove (iii) by unfolding the RHS, followed by substituting $\texttt{z}$ by $\texttt{t}$. All the above steps are summarized below, where LEFT-WEAKEN denotes the transformation above.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{True}}{(\texttt{x}\mapsto 0, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models (\texttt{x}\mapsto 0, \texttt{t}) * \mathsf{vlist}(\texttt{t})}\,(\textsc{substitution})}{(\texttt{x}\mapsto 0, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models (\texttt{x}\mapsto 0, \texttt{z}) * \mathsf{vlist}(\texttt{z})}\,(\textsc{right-unfold})}{(\texttt{x}\mapsto 0, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models \mathsf{vlist}(\texttt{x})\ (\text{iii})}\,(\textsc{left-weaken})}{(\texttt{x}\mapsto 0, \texttt{t}) * \mathsf{zero\_list}(\texttt{t}) \models \mathsf{vlist}(\texttt{x})\ (\text{ii})}$$

While the usefulness of having such a step is very clear, the conditions for its *correct* application is not obvious. To see this, let us use the same approach now but to prove that a list is also a list of zero elements, something that is clearly false. See Fig. 5.2. We proceed similarly as in the previous proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{True}}{(\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models (\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t})}\,(\textsc{substitution})}{(\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models (\texttt{x}\mapsto\_, \texttt{z}) * \mathsf{vlist}(\texttt{z})}\,(\textsc{right-unfold})}{(\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models \mathsf{vlist}(\texttt{x})\ (3)}\,(\textsc{right-strengthen})}{(\texttt{x}\mapsto\_, \texttt{t}) * \mathsf{vlist}(\texttt{t}) \models \mathsf{zero\_list}(\texttt{x})\ (2)}$$

Once again, we use the original proof obligation $\mathsf{vlist}(\texttt{t}) \models \mathsf{zero\_list}(\texttt{t})$ as an induction hypothesis, and this time, we transform the proof obligation (2) into (3): strengthen the RHS by replacing $\mathsf{zero\_list}(\texttt{x})$ with $\mathsf{vlist}(\texttt{x})$. Call this transformation RIGHT-STRENGTHEN. Clearly (3) is easily proven true, as shown.

This erroneous proof arises from a form of *circular reasoning*. Our challenge therefore is how to use induction correctly, as in Fig. 5.1, but avoid pitfalls such as in Fig. 5.2.

In this work, we propose a general proof method for recursive predicates that includes reasoning by induction. Our method is able to use *dynamically generated* formulas as induction hypotheses, and to enforce an *anti-circular* condition so that any application of an induction step is guaranteed to be correct. We shall see that our method is very different from that in traditional theorem proving systems where, after having chosen an induction tactic, the system will then search for appropriate induction variable(s) with a well-founded measure and appropriate induction hypotheses. In our framework, the predicates are defined by general recursive rules, without any explicit restriction to any well-founded orderings, and includes a domain of discourse that captures the mutable heap and properties of separation. More specifically:

- We automatically and efficiently *discharge* all commonly-used lemmas, extracted from a number of benchmarks used by other systems. These systems cannot automatically discharge such lemmas, but simply accept them as true facts.

- We demonstrate, in a different set of benchmarks in Section 5.7, that with our proof method, the common usage of lemmas can be *avoided*. This is because the properties of interest are covered by our method. In contrast, these properties cannot be discharged by the other systems without using lemmas.

  The impact of this is twofold. First, it means that for proving practical (but small) programs, the users are now free from the burden of providing custom user-defined lemmas. Second, it significantly boosts up the performance, since lemma applications, coupled with folding/unfolding, often induce a large search space.

- The proposed proof method gets us back the power of compositional reasoning in dealing with user-defined recursive predicates. While we have not been able to identify *precisely* the class where our proof method would be effective[1]; we do believe that its potential impact is huge. One important subclass that we can handle effectively is when both the antecedent and the consequent refer to the same structural shape but the antecedent simply makes a *stronger* statement about the values in the structure (e.g., to prove that a sorted list is also a list, an AVL tree is also a binary search tree, a list consists of all data values 999 is one that has all positive data, etc.).

[1] This is as *hard* as identifying the class where an invariant discovery technique guarantees to work.

In summary, we extend significantly the state-of-the-art proof methods, namely U+M based methods. We are able to prove relationships between general predicates of arbitrary arity, even when recursive definitions and the code are structurally dissimilar. In Section 5.2, we will motivate the need for our extension in more detail. Sections 5.3 and 5.4 contain the technical core. In Section 5.7, we evaluated our prototype implementation on a comprehensive set of benchmarks, including both academic algorithms and real programs. The benchmarks are collected from existing systems [Nguyen and Chin, 2008; Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Brotherston *et al.*, 2012], those considered as the state-of-the-art for the purpose of proving user-defined recursive data-structure properties in imperative languages. Section 5.8 discusses related work in detail and Section 5.9 concludes.

5.2 Motivation

In this Section, we motivate the need for inductive reasoning in proving user-defined recursive data-structure properties.

We first highlight scenarios, which are *ubiquitous* in realistic programs, and often lead to proof obligations involving unmatchable predicates. Later, we discuss the restriction of U+M paradigm in dealing with such proof obligations.

5.2.1 Scenario 1: Recursion Divergence

when the "recursion" in the recursive rules is structurally dissimilar to the program code.

This happens often with *iterative* programs and when the predicates are not *unary*, i.e., they relate two or more pointer variables, from which the program code traverse/manipulate the data structure in directions different from the definition.

```
elm = malloc()          assume(head!=null)
assume(tail!=null)      assume(head!=tail)
elm.next = null         elm = head
tail.next = elm         head = head.next
tail = elm              free(elm)
```

(a) Insert Tail (b) Remove Head

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \stackrel{def}{=} \mathtt{x}{=}\mathtt{y} \wedge \mathbf{emp} \;\mid\; \mathtt{x}{\neq}\mathtt{y} \wedge (\mathtt{x}{\mapsto}\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$$

(c) List Segment Definition

Figure 5.3: Implementation of a Queue

To illustrate, Fig. 5.3 shows the implementation of a queue using list segment, extracted from `OpenBSD/queue.h`, an open source program. Two operations of interest: (1) adding a new element into the end of a non-empty queue (`enqueue`, Fig. 5.3(a)); (2) deleting an element at the beginning of a non-empty queue (`dequeue`, Fig. 5.3(b)). A simple property we want to prove is that given a list segment representing a non-empty queue at the beginning, after each operation, we still get back a list segment.

In the two use cases, the "moving pointers" are necessary to recurse differently: the `tail` is moved in `enqueue` while the `head` is moved in `dequeue`. Consequently, no matter how we define list segments[2], where `head` and `tail` are the two pointers, at least one use case would recurse differently from the definition, thus exhibit the "recursion divergence" scenario and lead to a proof obligation involving unmatchable predicates. More concretely, if list segment is defined as in Fig. 5.3(c), the `enqueue` operation would lead to an obligation that is impossible for U+M to prove.

5.2.2 Scenario 2: Generalization of Predicate

> *when the predicate describing a loop invariant or a function needs to be used later to prove a weaker property.*

This happens in almost all realistic programs. The reason is because verification of functional correctness is performed *modularly*. More specifically, given the specifications for functions and invariants for loops, we can first perform *local* reasoning before composing the whole proof for the program using, in the context of Separation Logic, the *frame rule* [Reynolds, 2003]. It can be seen that, given such divide-and-conquer strategy, at the *boundaries* between local code fragments, we would need "generalization of predicate". A particularly important relationship between predicates, at the boundary point, is simply that one (the consequent) is *more general* than the other (the antecedent), representing a valid abstraction step.

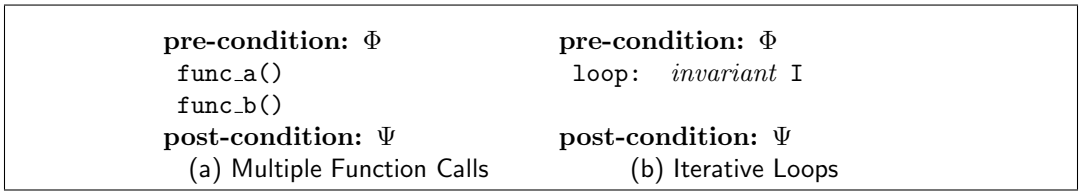


Figure 5.4: Modular Program Reasoning

[2]Typically, list segment can be defined in two ways: the moving pointer is either the *left* one or the *right* one.

Consider the boundaries between function calls, illustrated by the pattern in Fig. 5.4(a). We start with the pre-condition Φ, calling function `func_a` and then `func_b`. We then need to establish the post-condition Ψ. In traditional forward reasoning, we will write local (and consistent) specifications for `func_a` and `func_b` such that: (1) Φ is stronger than the pre-condition of `func_a`; (2) the post-condition of `func_a` is stronger than the pre-condition of `func_b`; (3) the post-condition of `func_b` is stronger than Ψ. It is hard, if not impossible, to ensure that for each pair (out of three) identified above, the antecedent and the consequent are constructed from matchable predicates. As a concrete example, in `bubblesort` program [Chin *et al.*, 2012], a boundary between two function calls requires us to prove that a *sorted* linked-list is also a linked-list.

We further argue that in software development, code reuse is often desired. The specification of a function, especially when it is a *library* function, should (or must) be relatively *independent* of the context where the function is plugged in. In each context, we might want to establish arbitrarily different properties, as long as they are *weaker* than what the function can guarantee. In such cases, it is almost certain that we will have proof obligations involving unmatchable predicates.

Now consider the boundaries caused by loops. In *iterative* algorithms, the loop invariants must be consistent with the code, and yet these invariants are only used later to prove a property often *not* specified using the identical predicates of the invariants. In the pattern shown by Fig. 5.4(b), this means that the proof obligations relating the pre-condition Φ to the invariant `I` and `I` to post-condition Ψ often involve unmatchable predicates. For example, programs manipulate lists usually have loops of which the invariants need to talk about list segments. Assume that (acyclic) linked-list is defined as below:

$$\begin{array}{rcl} \mathsf{list(x)} & \stackrel{def}{=} & \mathtt{x{=}null} \wedge \mathbf{emp} \\ & & \mid (\mathtt{x}{\mapsto}\mathtt{t}) * \mathsf{list(t)} \end{array}$$

Though $\widehat{\mathsf{ls}}$ and list are closely related, U+M can prove neither of the following obligations:

$$\widehat{\mathsf{ls}}(\mathtt{x}, \mathtt{null}) \models \mathsf{list}(\mathtt{x}) \tag{5.2.1}$$

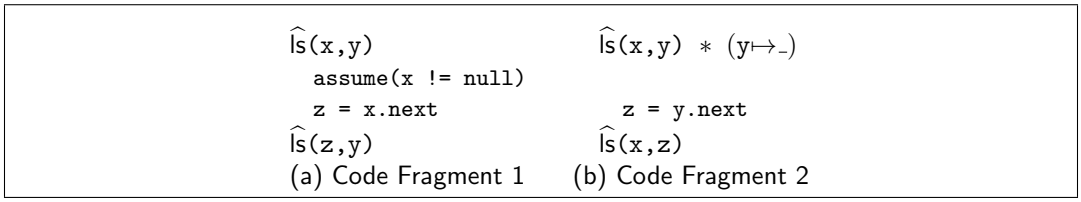
$$\widehat{\mathsf{ls}}(\mathtt{x}, \mathtt{y}) * \mathsf{list}(\mathtt{y}) \models \mathsf{list}(\mathtt{x}) \tag{5.2.2}$$

In summary, the above discussion connects to a serious issue in software development and verification: without the ability to relate predicates — when they are unmatchable — *compositional reasoning* is seriously hampered.

5.2.3 On Unfold-and-Match (U+M) Paradigm

As stated in Section 5.1, the dominating technique to manipulate user-defined recursive predicates is to employ the basic transformation steps of folding and unfolding the rules, together with formula abstraction, i.e., the U+M paradigm.

The main challenge of the U+M paradigm is clearly how to systematically search for such sequences of fold/unfold transformations. We believe recent works [Madhusudan *et al.*, 2012; Qiu *et al.*, 2013], we shall call the DRYAD works, have brought the U+M to a new level of automation. The key technical step is to use the *program statements* in order to *guide* the sequence of fold/unfold steps of the recursive rules which define the predicates of interest. For example, assume the definition for list segment $\widehat{\mathsf{ls}}$ in Fig. 5.3(c) and the code fragment in Fig. 5.5(a).



(a) Code Fragment 1 (b) Code Fragment 2

Figure 5.5: U+M with List Segments

Here we want to prove that given $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y})$ at the beginning, we should have $\widehat{\mathsf{ls}}(\mathtt{z},\mathtt{y})$ at the end. Since the code touches the "footprint" of `x` (second statement), it *directs* the unfolding of the predicate $\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y})$ containing `x`, to expose $\mathtt{x} \neq \mathtt{y} \wedge (\mathtt{x} \mapsto \mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$. The consequent can then be established via a simple *matching* from variable `z` to `t`.

Now we consider the code fragment in Fig. 5.5(b): instead of moving one position away from `x`, we move one away from `y`. To be convinced that U+M, however, cannot work, it suffices to see that unfolding/folding of $\widehat{\mathsf{ls}}$ does not change the *second argument* of the predicate $\widehat{\mathsf{ls}}$. Therefore, regardless of the unfolding/folding sequence, the arguments `y` on the LHS and `z` on the RHS would maintain and can never be matched satisfactorily.

The example in Fig. 5.5(b) exhibits the "recursion divergence" scenario mentioned above and ultimately is about relating two possible definitions of list segment (recursing either on the left or on the right pointer), which U+M fundamentally cannot handle. We will revisit this example in later Sections.

On Using Axioms and Lemmas: For systems that support general user-defined predicates [Chin *et al.*, 2012; Qiu *et al.*, 2013], they get around the limitation of U+M via the use,

without proof, of additional *user-provided* "lemmas" (the corresponding term used in [Qiu *et al.*, 2013] is "axioms"). As a matter of fact, in the viewpoint of proof method, it is unacceptable that in order to prove more programs, we *continually* add in more custom lemmas to facilitate the proof system.

5.3 The Assertion Language $CLP(\mathcal{H})$

The explicit naming of heaps has emerged naturally in several extensions of Separation Logic (SL) as an aid to practical program verification. Reynolds conjectured that referring explicitly to the current heap in specifications would allow better handles on data structures with sharing [Reynolds, 2003]. In this vein, [Duck *et al.*, 2013] extends Hoare Logic with explicit heaps. This extension allows for strongest post conditions, and is therefore suitable for "practical program verification" [Brotherston and Villard, 2014] via constraint-based symbolic execution.

In this work, we start with the existing specification language in [Duck *et al.*, 2013], which has two notable features: (a) the use of explicit heap variables, and (b) user-defined recursive properties in a wrapper logic language based on recursive rules. The language provides a new level of expressiveness for specifying properties of heap-manipulating programs. We remark that, common specifications written in traditional Separation Logic, can be automatically compiled into this language.

We will be brief here and refer interested readers to [Duck *et al.*, 2013] for more details. A *heap* is a *finite partial map* from *positive* integers to integers, i.e., $\mathsf{Heaps} = \mathbb{Z}_+ \rightharpoonup_{\text{fin}} \mathbb{Z}$. Given a heap $h \in \mathsf{Heaps}$ with domain $D = dom(h)$, we sometimes treat h as the set of pairs $\{(p, v) \mid p \in D \wedge v = h(p)\}$. We note that when a pair (p, v) belongs to some heap h, it is necessary that p is not `null` ($p \neq 0$). The $\mathcal{H}$-language is the first-order language over heaps.

We use $(*)$ and $(\doteq)$ operators to respectively denote heap disjointness and equation. Intuitively, a constraint like $H \doteq H_1 * H_2$ restricts H_1 and H_2 to be disjoint while giving a name H to the conjoined heaps $H_1 * H_2$.

As in [Duck *et al.*, 2013], $\mathcal{H}$ is then extended with *user-defined* recursive predicates. We use the framework of *Constraint Logic Programming* (CLP) [Jaffar and Maher, 1994] to inherit its syntax, semantics, and most importantly, its built-in notions of unfolding rules. For brevity, we just informally explain the language. The following rules constitutes

a recursive definition of predicate $\mathsf{list}(x, L)$, which specifies a skeleton *list*.

$\mathsf{list}(x, L)$:- $x = 0$, $L \doteq \Omega$.
$\mathsf{list}(x, L)$:- $L \doteq (x{\mapsto}t){*}L_1$, $\mathsf{list}(t, L_1)$.

The *semantics* of a set of rules is traditionally known as the "least model" semantics (LMS). Essentially, this is the set of groundings of the predicates which are true when the rules are read as traditional implications. The rules above dictates that all true groundings of $\mathsf{list}(x, L)$ are such that x is an integer, L is a heap which contains a skeleton list starting from x. More specifically, when the list is empty, the root node is equal to `null` ($x = 0$), and the heap is empty ($L \doteq \Omega$). Otherwise, we can split the heap L into two disjoint parts: a singleton heap ($x{\mapsto}t$) and the remaining heap L_1, where L_1 corresponds to the heap that contains a skeleton list starting from t.

We now provide the definitions for list segments, which will be used in our later examples. Do note the extra explicit heap variable L, in comparison with corresponding definitions in SL.

$\widehat{\mathsf{ls}}(x, y, L)$:- $x{=}y$, $L \doteq \Omega$.
$\widehat{\mathsf{ls}}(x, y, L)$:- $x{\neq}y$, $L \doteq (x{\mapsto}t){*}L_1$, $\widehat{\mathsf{ls}}(t, y, L_1)$.

$\mathsf{ls}(x, y, L)$:- $x{=}y$, $L \doteq \Omega$.
$\mathsf{ls}(x, y, L)$:- $x{\neq}y$, $L \doteq (t{\mapsto}y){*}L_1$, $\mathsf{ls}(x, t, L_1)$.

We also emphasize that the main advantage of this language is the possibility of deriving the strongest postcondition along each program path. It is indeed the main contribution of [Duck *et al.*, 2013]. Specifically, in order to prove the Hoare triple $\{\phi\}S\{\psi\}$ for a loop-free program S, we simply generate strongest postcondition ψ' along each of its straight-line paths and obtain the verification condition $\psi' \models \psi$. Note that the handling of loops can be reduced to this loop-free setting because of user-specified invariants. For procedure calls, we still make use of the (standard) frame rule to generate proof obligations. We put forward that, in all our experiments (Section 5.7), the verification conditions are generated using the frame rule (manually though) and the symbolic execution rules of [Duck *et al.*, 2013].

5.4 The Proof Method

Background on CLP: This is provided for the convenience of the readers. An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ is a tuple of $\mathcal{H}$ terms. A *rule*

is of the form $A\texttt{:-}\Psi, \tilde{B}$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A finite set of rules is then used to define a predicate. A *goal* has exactly the same format as the body of a rule. A goal that contains only constraints and no atoms is called *final*.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each variable into its intended universe of discourse: an integer or a heap, in the case of our CLP($\mathcal{H}$). Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $\mathcal{G} \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \Psi)$ and P denote a non-final goal and a set of rules respectively. Let $R \equiv A\texttt{:-}\ \Psi_1, C_1, \cdots, C_m$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a clause R, denoted $reduct(\mathcal{G}, R)$, is of the form
$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A, \Psi, \Psi_1)$
provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$, where $\mathcal{G}_i,\ i > 0$ is a reduct of $\mathcal{G}_{i-1}$. A *derivation tree* for a goal is defined in the obvious way.

Definition 1 (Unfold)**.** *Given a program P and a goal $\mathcal{G}$:* $\textsc{unfold}(\mathcal{G})$ *is* $\{\mathcal{G}' | \exists R \in P : \mathcal{G}' = reduct(\mathcal{G}, R)\}$. □

Given a goal $\mathcal{L}$ and an atom $p \in \mathcal{L}$, $\textsc{unfold}_p(\mathcal{L})$ denotes the set of formulas transformed from $\mathcal{L}$ by unfolding p.

Definition 2 (Entailment)**.** *An* entailment *is of the form $\mathcal{L} \models \mathcal{R}$, where $\mathcal{L}$ and $\mathcal{R}$ are goals.* □

This work considers proving the validity of the entailment $\mathcal{L} \models \mathcal{R}$ under a given program P. This entailment means that $lm(P) \models (\mathcal{L} \rightarrow \mathcal{R})$, where $lm(P)$ denotes the "least model" of the program P which defines the recursive predicates — called *assertion* predicates — occurring in $\mathcal{L}$ and $\mathcal{R}$. This is simply the set of all groundings of atoms of the assertion

predicates which are *true* in P. The expression $(\mathcal{L} \rightarrow \mathcal{R})$ means that, for each grounding θ of $\mathcal{L}$ and $\mathcal{R}$, $\mathcal{L}\theta$ is in $lm(P)$ implies that so is $\mathcal{R}\theta$.

5.4.1 Unfold and Match (U+M)

Assume that we start off with $\mathcal{L} \models \mathcal{R}$. If this entailment can be proved directly, by unification and/or consulting an off-the-shelf SMT solver, we say that the entailment is trivial: a *direct proof* is obtained even without considering the "meaning" of the recursively defined predicates (they are treated as *uninterpreted*). When it is not the case — the entailment is non-trivial — a standard approach is to apply unfolding/folding until all the "frontier" become trivial. We note that, in our framework, we perform only unfolding, but now to both the LHS (the antecedent) and the RHS (the consequent) of the entailment. The effect of unfolding the RHS is similar to a folding operation on the LHS. In more detail, when direct proof fails, U+M paradigm proceeds in two possible ways:

- First, select a recursive atom $p \in \mathcal{L}$, unfold $\mathcal{L}$ wrt. p and obtain the goals $\mathcal{L}_1, \ldots, \mathcal{L}_n$. The validity of the original entailment can now be obtained by ensuring the validity of *all* the entailments $\mathcal{L}_i \models \mathcal{R}$ $(1 \leq i \leq n)$.
- Second, select a recursive atom $q \in \mathcal{R}$, unfold $\mathcal{R}$ wrt. q and obtain the goals $\mathcal{R}_1, \ldots, \mathcal{R}_m$. The validity of the original entailment can now be obtained by ensuring the validity of *any one of* the entailments $\mathcal{L} \models \mathcal{R}_j$ $(1 \leq j \leq m)$.

So the proof process can proceed recursively either by proving *all* $\mathcal{L}_i \models \mathcal{R}$ or by proving *one* $\mathcal{L} \models \mathcal{R}_j$ for some j. Since the original LHS and RHS usually contain more than one recursive atoms, this proof process naturally triggers a search tree. Termination can be guaranteed by simply bounding the maximum number of left and right unfolds allowed. In practice, the number of recursive atoms used in an entailment is usually small, thus resulting tree size is often manageable.

5.4.2 Formula Re-writing with Dynamic Induction Hypotheses

We now present a formal calculus for the proof of $\mathcal{L} \models \mathcal{R}$ that goes beyond unfold-and-match. The power of our proof framework comes from the key concept: *induction*.

Definition 3 (Proof Obligation)**.** *A* proof obligation *is of the form* $\tilde{A} \vdash \mathcal{L} \models \mathcal{R}$ *where* $\mathcal{L}$ *and* $\mathcal{R}$ *are goals and* $\tilde{A}$ *is a set of pairs* $\langle A; p \rangle$*, where* A *is an* assumed *entailment and* p *is a* recursive atom. □

$$\text{(CP)}\quad \frac{\texttt{True}}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \qquad \mathcal{L} \models_{\texttt{SMT}} \mathcal{R}\text{, where recursive atoms are treated as uninterpreted}$$

$$\text{(SUB)}\quad \frac{\tilde{A} \vdash \mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R}\theta}{\tilde{A} \vdash \mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R} \wedge p(\tilde{y})} \qquad \begin{array}{l}\text{there exists a substitution } \theta \text{ for}\\ \text{existential variables in } \tilde{y} \text{ s.t. } \mathcal{L} \wedge p(\tilde{x}) \models_{\texttt{SMT}} \tilde{x} = \tilde{y}\theta\end{array}$$

$$\text{(LU+I)}\quad \frac{\bigcup_{i=1}^{n} \{\tilde{A} \cup \{\langle \mathcal{L} \models \mathcal{R}; p\rangle\} \vdash \mathcal{L}_i \models \mathcal{R}\}}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \qquad \begin{array}{l}\text{Select an atom } p \in \mathcal{L} \text{ and}\\ \textsc{unfold}_p(\mathcal{L}) = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}\end{array}$$

$$\text{(RU)}\quad \frac{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}'}{\tilde{A} \vdash \mathcal{L} \models \mathcal{R}} \qquad \begin{array}{l}\text{Select an atom } q \in \mathcal{R} \text{ and}\\ \mathcal{R}' \in \textsc{unfold}_q(\mathcal{R})\end{array}$$

$$\text{(IA-1)}\quad \frac{\tilde{A} \vdash \mathcal{R}'\theta \wedge \mathcal{L}_2 \models \mathcal{R}}{\tilde{A} \vdash p(\tilde{x}) \wedge \mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}} \qquad \begin{array}{l}\langle p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'; p(\tilde{y})\rangle \in \tilde{A} \text{ and } \texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y})),\\ \text{there exists a renaming } \theta \text{ s.t. } \tilde{x} = \tilde{y}\theta \text{ and } \mathcal{L}_1 \models_{\texttt{DP}} \mathcal{L}'\theta\end{array}$$

$$\text{(IA-2)}\quad \frac{\tilde{A} \vdash \mathcal{L}_1 \models \mathcal{L}'\theta}{\tilde{A} \vdash p(\tilde{x}) \wedge \mathcal{L}_1 \models \mathcal{R}} \qquad \begin{array}{l}\langle p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'; p(\tilde{y})\rangle \in \tilde{A} \text{ and } \texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y}))\\ \text{and there exists a renaming } \theta \text{ s.t. } \tilde{x} = \tilde{y}\theta \text{ and } \mathcal{R}'\theta \models_{\texttt{DP}} \mathcal{R}\end{array}$$

Figure 5.6: General Proof Rules

The role of proof obligations is to capture the state of the proof process. Each element in $\tilde{A}$ is a pair, of which the first is an entailment A whose truth can be assumed inductively. A acts as an (dynamically generated) induction hypothesis and can be used to transform subsequently encountered obligations in the proof path. The second is a recursive atom p, to which the application of a left unfold gives rise to the addition of the induction hypothesis A.

Our proof rules – the obligation at the bottom, and its reduced form on top – are presented in Fig. 5.6. Given $\mathcal{L} \models \mathcal{R}$, our proof shall start with $\emptyset \vdash \mathcal{L} \models \mathcal{R}$, and proceed by repeatedly applying these rules. Each rule operates on a proof obligation. In this process, the proof obligation may be discharged (indicated by `True`); or new proof obligation(s) may be produced. $\mathcal{L} \models_{\texttt{SMT}} \mathcal{R}$ denotes the validity of $\mathcal{L} \models \mathcal{R}$ is obtained by consulting a generic SMT solver.

- The *substitution* (SUB) rule removes one occurrence of an assertion predicate, say atom $p(\tilde{y})$, appearing in the RHS of a proof obligation. Applying the (SUB) rule repeatedly will ultimately reduce a proof obligation to the form which contains no recursive atoms in the RHS, while at the same time (hopefully) most existential variables on the RHS are eliminated. Then, the *constraint proof* (CP) rule may be attempted by simply treating all remaining recursive atoms (in the LHS) as uninterpreted and by applying the underlying

theory solver assumed in the language we use.

The combination of (SUB) and (CP) rules attempts, what we call, a *direct proof*. In principle, it is similar to the process of "matching" in the U+M paradigm. For brevity we then use $\mathcal{L} \models_{\mathtt{DP}} \mathcal{R}$ to denote the fact that the validity of $\mathcal{L} \models \mathcal{R}$ can be proved directly using only (SUB) and (CP) rules.

• The *left unfold with induction hypothesis* (LU+I) is a key rule. It selects a recursive atom p on the LHS and performs a complete unfold of the LHS wrt. the atom p, producing a new set of proof obligations. The original obligation, while being removed, is added as an assumption to every newly produced proof obligation, opening the door for the later being used as an induction hypothesis. For technical reason needed below, we do not just add the obligation $\mathcal{L} \models \mathcal{R}$ as an assumption, but also need to keep track of the atom p. This is why in the rule we see a pair $\langle \mathcal{L} \models \mathcal{R}; p \rangle$ added into the current set of assumptions $\tilde{A}$.

On the other hand, the *right unfold* (RU) rule selects some recursive atom q and performs an unfold on the RHS of a proof obligation wrt. q. In the proof process, the two unfold rules will be systematically interleaved.

Example 4. *Consider the following proof obligation:*

$$\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2 \,.$$

$$\cfrac{\cfrac{\cfrac{\mathtt{True}}{\tilde{A} \vdash \mathsf{list}(x, L) \models x = x, L_1 \doteq \Omega, L \doteq L_1 * L}\,(\text{CP})}{\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, x, L_1), L \doteq L_1 * L}\,(\text{RU})}{\tilde{A} \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2}\,(\text{SUB})$$

Figure 5.7: Proving with just U+M

In Fig. 5.7, we show how this proof obligation can be successfully dispensed by applying (SUB), (RU), and (CP) rules in sequence. Note how the (SUB) rule binds the existential variable y to x, simplifying the RHS of the proof obligation.

• The *induction applications*, namely (IA-1) and (IA-2) rules, transform the current obligation by making use of an assumption which has been added by the (LU+I) rule. The two rules, also called the "induction rules" for short, allow us to treat previously encountered obligations as possible induction hypotheses.

Instead of directly proving the current obligation $\mathcal{L} \models \mathcal{R}$, we now proceed by finding $\overline{\mathcal{L}}$ and $\overline{\mathcal{R}}$ such that $\mathcal{L} \models \overline{\mathcal{L}} \models \overline{\mathcal{R}} \models \mathcal{R}$. The key here is to find those candidate goals where

the validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ directly follows from a "similar" assumption A, together with θ to rename all the variables in A to the variables in the current obligation, namely $\mathcal{L} \models \mathcal{R}$. Assumption A is an obligation which has been previously encountered in the proof process, and $A\theta$ assumed to be true, as an induction hypothesis. Particularly, we choose $\overline{\mathcal{L}}$ and $\overline{\mathcal{R}}$ so we can (easily) find a renaming θ such that $A\theta \implies \overline{\mathcal{L}} \models \overline{\mathcal{R}}$ ($\implies$ denotes logical implication).

To be more deterministic and to prevent us from transforming to obligations harder than the original one, we require that at least one of the remaining two entailments, namely $\mathcal{L} \models \overline{\mathcal{L}}$ and $\overline{\mathcal{R}} \models \mathcal{R}$, is discharged quickly by a direct proof.

In (IA-1) rule, given the current obligation $p(\tilde{x}) \wedge \mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}$ and an assumption $A \equiv p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'$, we choose $p(\tilde{x}) \wedge \mathcal{L}'\theta \wedge \mathcal{L}_2$ to be our $\overline{\mathcal{L}}$ and $\mathcal{R}'\theta \wedge \mathcal{L}_2$ to be our $\overline{\mathcal{R}}$. We can see that the validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ directly follows from the assumption $A\theta$. One restriction onto the renaming θ, to avoid circular reasoning, is that θ must rename $\tilde{y}$ to $\tilde{x}$ where $p(\tilde{x})$ is an atom which has been generated after $p(\tilde{y})$ had been unfolded. Such fact is indicated by $\texttt{gen}(p(\tilde{x})) \geq \texttt{kill}(p(\tilde{y}))$ in our rule. While $\texttt{gen}(p)$ denotes the timestamp when the recursive atom p is generated during the proof process, $\texttt{kill}(p)$ denotes the timestamp when p is unfolded and removed. Another side condition for this rule is that the validity of $\mathcal{L} \models \overline{\mathcal{L}}$, or equivalently, $\mathcal{L}_1 \models \mathcal{L}'\theta$ is discharged immediately by a direct proof.

In (IA-2) rule, given the current obligation $p(\tilde{x}) \wedge \mathcal{L}_1 \models \mathcal{R}$ and an assumption $A \equiv p(\tilde{y}) \wedge \mathcal{L}' \models \mathcal{R}'$, on the other hand, $p(\tilde{y})\theta \wedge \mathcal{L}'\theta$ serves as our $\overline{\mathcal{L}}$ while $\mathcal{R}'\theta$ serves as our $\overline{\mathcal{R}}$. The validity of $\overline{\mathcal{L}} \models \overline{\mathcal{R}}$ trivially follows from the assumption $A\theta$, namely $p(\tilde{x}) \wedge \mathcal{L}'\theta \models \mathcal{R}'\theta$. As in (IA-1), we also put similar restriction upon the renaming θ. Another side condition we require is that the validity of $\overline{\mathcal{R}} \models \mathcal{R}$ can be discharged immediately by a direct proof. At this point we could see the duality nature of (IA-1) and (IA-2).

Now let us briefly and intuitively explain the restriction upon the renaming θ. Here we make sure that θ renames atom $p(\tilde{y})$ to atom $p(\tilde{x})$, where $p(\tilde{x})$ has been generated after $p(\tilde{y})$ had been unfolded (and removed). This helps to rule out certain potential θ which does not correspond to a number of left unfolds. Such restriction helps ensure *progressiveness* in the proof process before the induction rules can take place. Otherwise, assuming the truth of $A\theta$ in constructing the proof for A might not be valid. This is the reason why for each element of $\tilde{A}$, we not only keep track of the assumption, but also the recursive atom p to which the application of (LU+I) gives rise to the addition of such assumption.

It is important to note that, our framework as it stands, does not require any consideration of a base case, nor any well-founded measure. Instead, we depend on the Least Model Semantics (LMS) of our assertion language and the above-mentioned restrictions on the renaming θ. In other words, by constraining the use of the rules, which is transparent to the user, we guarantee to achieve a well-founded conclusion.

Least Model Semantics: Let us now give an example to illustrate why our proof is working under the LMS. Consider the recursive predicate $\mathtt{p}$, defined as

$$\mathtt{p}(x) \texttt{ :- } \mathtt{p}(x).$$

and the following two proof obligations:

$$\mathtt{p}(x) \models \mathsf{list}(x, L) \tag{5.4.1}$$

$$\mathsf{list}(x, L) \models \mathtt{p}(x) \tag{5.4.2}$$

We will now demonstrate that our method can prove (5.4.1), but not (5.4.2). We remark that (5.4.1) holds because under the LMS, the LHS has no model; therefore no refutation can be found regardless of what the RHS is. In other words, *false* implies anything. On the other hand, (5.4.2) does not hold because $x = 0$ (and $L \doteq \Omega$) is a model of the LHS, but not a model of the RHS.

$$\cfrac{\cfrac{\cfrac{\texttt{True}}{\{A\} \vdash \mathsf{list}(x, L) \models \mathsf{list}(x, L)}\,(\textsc{cp})}{\{A\} \vdash \mathtt{p}(x) \models \mathsf{list}(x, L)}\,(\textsc{ia-1})}{\emptyset \vdash \mathtt{p}(x) \models \mathsf{list}(x, L)}\,(\textsc{lu+i})$$

Figure 5.8: Our Proof for (5.4.1)

$$\cfrac{\cfrac{\{A'\} \vdash x = 0, L \doteq \Omega \models \mathtt{p}(x) \quad \vdots}{\emptyset \vdash \mathsf{list}(x, L) \models \mathtt{p}(x)}\,(\textsc{lu+i})}{\emptyset \vdash \mathsf{list}(x, L) \models \mathtt{p}(x)}\,(\textsc{ru})$$

Figure 5.9: An Unsuccessful Attempt for (5.4.2)

Fig. 5.8 shows how our method would handle (5.4.1). We first perform a left unfolding, adding $A \equiv \langle \mathtt{p}(x) \models \mathsf{list}(x, L); \mathtt{p}(x) \rangle$ into the set of assumptions. Note that this unfolding step kills the predicate $\mathtt{p}(x)$ and generates a new predicate $\mathtt{p}(x)$. Thus the rule (IA-1) is

applicable now. We then re-write the LHS from $\mathsf{p}(x)$ to $\mathsf{list}(x, L)$. Finally the proof succeeds by consulting constraint solver, treating $\mathsf{list}(x, L)$ as uninterpreted.

In contrast, now consider obligation (5.4.2) in Fig. 5.9. Obviously, a direct proof for this is not successful. However, if we proceed by a right unfold first, we get back the same obligation. Different from before, and importantly, now no new assumption is added. We can see that the step does not help us progress and therefore performing right unfold repetitively would get us nowhere. Now consider performing a left unfold on the obligation. The proof succeeds if we can discharge both

$$\{A'\} \vdash x = 0, L \doteq \Omega \models p(x) \;\; \texttt{and}$$
$$\{A'\} \vdash L \doteq (x{\mapsto}t){*}L_1, \mathsf{list}(t, L_1) \models p(x)\,,$$

where $A' \equiv \langle \mathsf{list}(x, L) \models \mathsf{p}(x); \mathsf{list}(x, L) \rangle$.

Focus on the obligation $\{A'\} \vdash x = 0, L \doteq \Omega \models p(x)$. Clearly consulting a constraint solver or performing substitution does not help. Rule (LU+I) is not applicable since no recursive predicate on the LHS. As before, we cannot progress using (RU) rule. Importantly, the side conditions prevent (IA-1) and (IA-2) from taking place. In summary, with our proof rules, this (wrong fact) cannot be established.

5.4.3 Proving the Two Motivating Examples

Let us now revisit the two motivating examples introduced earlier, on which both U+M and "Cyclic Proof" are not effective. The main reason is that both examples involve unmatchable predicates while at the same time exhibiting "recursion divergence".

Example 5. *Consider the entailment relating two definitions of list segments:* $\widehat{\mathsf{ls}}(x, y, L) \models \mathsf{ls}(x, y, L)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{True}}\,(\text{CP})}{\{A_1, A_2\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \doteq (z{\mapsto}y){*}L_2 \models x{\neq}y, L \doteq (z{\mapsto}y){*}(x{\mapsto}t){*}L_2}\,(\text{SUB})}{\{A_1, A_2\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \doteq (z{\mapsto}y){*}L_2, \mathsf{ls}(x, z, (x{\mapsto}t){*}L_2) \models x{\neq}y, L \doteq (z_1{\mapsto}y){*}L_3, \mathsf{ls}(x, z_1, L_3)}\,(\text{IA-1})}{\{A_1, A_2\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \doteq (z{\mapsto}y){*}L_2, \mathsf{ls}(t, z, L_2) \models x{\neq}y, L \doteq (z_1{\mapsto}y){*}L_3, \mathsf{ls}(x, z_1, L_3)}\,(\text{RU})}{\{A_1, A_2\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, t{\neq}y, L_1 \doteq (z{\mapsto}y){*}L_2, \mathsf{ls}(t, z, L_2) \models \mathsf{ls}(x, y, L)}\,(\text{LU+I})}{\{A_1\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, \mathsf{ls}(t, y, L_1) \models \mathsf{ls}(x, y, L)}\,(\text{IA-1})}{\{A_1\} \vdash x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, \widehat{\mathsf{ls}}(t, y, L_1) \models \mathsf{ls}(x, y, L)}\,(\text{LU+I})}{\emptyset \vdash \widehat{\mathsf{ls}}(x, y, L) \models \mathsf{ls}(x, y, L)}$$

where $A_1 \equiv \langle \widehat{\mathsf{ls}}(x, y, L) \models \mathsf{ls}(x, y, L); \widehat{\mathsf{ls}}(x, y, L) \rangle$ **and** $A_2 \equiv \langle x{\neq}y, L \doteq (x{\mapsto}t){*}L_1, \mathsf{ls}(t, y, L_1) \models \mathsf{ls}(x, y, L); \mathsf{ls}(t, y, L_1) \rangle$

Figure 5.10: Proving $\widehat{\mathsf{ls}}(x, y, L) \models \mathsf{ls}(x, y, L)$.

Our method can discharge this obligation by applying (IA-1) rule twice. For space reason, in Fig. 5.10, we only show the interesting path of the proof tree (leftmost position). First,

we unfold the predicate $\widehat{\mathsf{ls}}(x, y, L)$ in the LHS of the given obligation via (LU+I) rule. The original obligation, while being removed, is added as an assumption A_1. We next make use of A_1 as an induction hypothesis to perform a re-writing step, i.e., an application of (IA-1) rule. Similarly, in the third step, we unfold the predicate $\mathsf{ls}(t, y, L_1)$ in the LHS via (LU+I) rule and add the assumption A_2. After unfolding in the RHS via (RU) rule and re-writing with the induction hypothesis A_2 using (IA-1) rule, we are able to bind the existential variable z_1 to z and simplify both sides of the proof obligation using (SUB) rule. Finally, the proof path is terminated by consulting a constraint solver, i.e., using (CP) rule.

Example 6. *Consider the entailment:*

$$\mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2.$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{True}}{\{A\} \vdash x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{list}(y, L_2), L_1{*}L_2 \models L_4 \doteq (t{\mapsto}y){*}L_2, L_1{*}L_2 \doteq L_3{*}L_4}\,(\text{CP})}{\{A\} \vdash x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{list}(y, L_2), L_1{*}L_2 \models L_4 \doteq (t{\mapsto}y_1){*}L_5, \mathsf{list}(y_1, L_5), L_1{*}L_2 \doteq L_3{*}L_4}\,(\text{SUB})}{\{A\} \vdash x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(t, L_4), L_1{*}L_2 \doteq L_3{*}L_4}\,(\text{RU})}{\{A\} \vdash x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{ls}(x, t, L_3), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2}\,(\text{IA-2})}{\emptyset \vdash \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2}\,(\text{LU+I})\ \vdots$$

where $A \equiv \langle \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2; \mathsf{ls}(x, y, L_1)\rangle$

Figure 5.11: Proving $\mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L_1{*}L_2 \models \mathsf{list}(x, L), L \doteq L_1{*}L_2$.

Fig. 5.11 shows, only the interesting proof path, how we can successfully prove this entailment using the (IA-2) rule. We first unfold $\mathsf{ls}(x, y, L_1)$ in the LHS, adding A into the set of assumptions. Then using A as an induction hypothesis, we can rewrite the current obligation via (IA-2) rule. Note that, here we use (IA-2) rule instead of (IA-1) rule as in previous example. After applying (RU) rule, we are able to bind the existential variable y_1 to y and simplify both sides of the proof obligation with (SUB) rule. Finally, the proof path is terminated by consulting a constraint solver, i.e., using (CP) rule.

Let us pay a closer attention at the step where we attempt re-writing, making using the available induction hypothesis. For the sake of discussion, assume that instead of (IA-2) we now attempt to apply rule (IA-1). The requirement for θ forces it to rename x to x and y to t. However, the side condition $\mathcal{L}_1 \models_{\mathtt{DP}} \mathcal{L}'\theta$ cannot be fulfilled, since

$$x \neq y, L_1 \doteq (t{\mapsto}y){*}L_3, \mathsf{list}(y, L_2), L_1{*}L_2 \not\models_{\mathtt{DP}} \mathsf{list}(t, \_).$$

Now return to the attempt of (IA-2) rule. The RHS of the current obligation matches with the RHS of the *only* induction hypothesis perfectly. This matching requires θ to rename x

back to x. On the LHS, we further require θ to rename y to t so that $\mathsf{ls}(x,t) \equiv \mathsf{ls}(x,y)\theta$. Note that $\mathsf{ls}(x,t)$ was indeed generated after $\mathsf{ls}(x,y)$ had been unfolded and removed (i.e., killed). The remaining transformation is more straightforward.

5.5 Implementation

Let us briefly highlight our implementation, which intuitively follows from the proof rules in Sec. 5.4. The main algorithm is in Figure 5.13. In the figure, we use $X \cup\!= Y$ to denote $X := X \cup Y$.

We start off by calling the function Prove with the original obligation $\mathcal{L} \models \mathcal{R}$, the set of assumptions $\tilde{A}$ to be $\emptyset$, and all the counters lb, rb, ib to be 0. The counters lb, rb, ib are to keep track of, respectively, how many left unfolds, right unfolds, and inductions have been applied in this current path. These counters are to ensure that our algorithm terminates. In our experiments, the typical values for `INDUCTIONBOUND, MAXLEFTBOUND, MAXRIGHTBOUND` are respectively 3, 5, 5.

Typically an unoptimized proof obligation usually can be partitioned into a number of smaller and simpler proof obligations (e.g., by eliminating redundant terms and variables). This step can be implemented using any standard proof slicing technique and is not the focus of our discussion.

Base Case: The function DirectProof acts as the base case of our recursive algorithm. For each proof obligation, we first attempt a *direct proof*, i.e., to discharge by applying rule (SUB) repetitively and then querying Z3 solver [De Moura and Bjørner, 2008a], after treating all recursive predicates in the LHS as uninterpreted, as in (CP)-rule.

Intuitively, this step succeeds if the proof obligation is simple "enough" such that a *proof by matching* can be achieved. We note here that, our proof rules in Section 5.4 allow other rules, e.g., (RU) rule in Example 4, to interleave with the (SUB) and (CP) rules. However, in our deterministic implementation, applications of (SUB) and (CP) rules are coupled together.

Let us examine the function DirectProof. If there is a recursive predicate q on the RHS, but not in the LHS, the function returns immediately, indicating failure with $\bot$. Otherwise, the function then proceeds by finding some (not exhaustive) substitutions Θ such that with each $\theta \in \Theta$, we can simultaneously remove all the recursive predicates on the RHS. This process will remove most existential variables on the RHS, since existential variables usually appear in some recursive predicates.

In case there remain some existential variables on the RHS, we attempt to bind them with the obvious candidates on the LHS (therefore extend θ to θ'). After this attempt, if the RHS contains no existential variables, we then call an SMT solver for entailment check. If the answer is yes, θ' is returned, indicating that a direct proof has been achieved.

```
function Prove(L ⊨ R, Ã, lb, rb, ib)
      /* Natural proof, i.e. by unification and SMT */
⟨1⟩   if (DirectProof(L ⊨ R)) return true

      let L = Φ, p_1, ..., p_n and R = Ψ, q_1, ..., q_m
⟨2⟩   OrSet := ∅
⟨3⟩   if (ib < INDUCTIONBOUND)        /* Induction Application */
⟨4⟩      foreach(⟨p(x̃) ∧ L' ⊨ R'); p(ỹ)⟩ ∈ Ã)
⟨5⟩         Find p(x̃) ∈ L s.t. gen(p(x̃)) ≥ kill(p(ỹ))
⟨6⟩         Find L_1 ⊆ L s.t. θ := DirectProof(L_1 ⊨ L') ≠ ⊥
⟨7⟩         if(x̃ = ỹθ and θ is a valid renaming)
⟨8⟩            L_new := L \ {p(x̃)} \ L_1 ∪ R'θ
⟨9⟩            OrSet ∪= {⟨L_new ⊨ R, Ã, lb, rb, ib + 1⟩}
⟨10⟩     foreach(⟨p(ỹ) ∧ L' ⊨ R'); p(ỹ)⟩ ∈ Ã)
⟨11⟩        if (θ_1 := DirectProof(R' ⊨ R) = ⊥) continue
⟨12⟩        Find p(x̃) ∈ L s.t. gen(p(x̃)) ≥ kill(p(ỹ))
⟨13⟩        Find a valid renaming θ ⊇ θ_1 s.t. x̃ = ỹθ
⟨14⟩        OrSet ∪= {⟨L \ {p(x̃)} ⊨ L'θ, Ã, lb, rb, ib + 1⟩}

⟨15⟩  if (lb < MAXLEFTBOUND)        /* Left Unfold */
⟨16⟩     foreach (p_i ∈ L)
⟨17⟩        Obs := ∅
⟨18⟩        Ã' := Ã ∪ {⟨L ⊨ R; p_i⟩}
⟨19⟩        foreach (L_j ∈ ({L_1, L_2, ... L_l} := UNFOLD(p_i)))
⟨20⟩           ob := ⟨(L_j ∪ L \ {p_i}) ⊨ R, Ã', lb + 1, rb, ib⟩
⟨21⟩           if (trivially_true(ob)) continue
⟨22⟩           Obs := Obs ∪ {ob}
⟨23⟩        if (Obs = ∅) return true else OrSet ∪= {Obs}

⟨24⟩  if (rb < MAXRIGHTBOUND and ¬contradict(L ⊨ R))
         /* Right Unfold */
⟨25⟩     foreach (q_i ∈ R)
⟨26⟩        foreach( R_j ∈ {R_1, R_2, ... R_k} := UNFOLD(q_i))
⟨27⟩           ob = ⟨L ⊨ (R_j ∪ R \ {q_i}), Ã, lb, rb + 1, ib⟩
⟨28⟩           OrSet ∪= {{ob}}

⟨29⟩  if (OrSet = ∅) return false
⟨30⟩  OrSet := OrderByHeuristics(OrSet)
⟨31⟩  foreach (Obs ∈ OrSet)
⟨32⟩     if (ProveAll(Obs)) return true
⟨33⟩  return false
endfunction
```

Figure 5.12: The Main Algorithm

```
function ProveAll(Obs)
⟨34⟩  foreach (⟨L ⊨ R, Ã, lb, rb, ib⟩ ∈ Obs)
⟨35⟩      if (¬ Prove(L ⊨ R, Ã, lb, rb, ib)) return false;
endfunction

function DirectProof(L ⊨ R)
⟨36⟩  if (∃ q(x̃) ∈ R such that ∄ q(ỹ) ∈ L) return ⊥
⟨37⟩  L' := get_all_recursive(L)
⟨38⟩  R' := get_all_recursive(R)
⟨39⟩  Θ := {substitution θ | R'θ ⊆ L'}
⟨40⟩  if (Θ = ∅) return ⊥
⟨41⟩  foreach (θ ∈ Θ)
⟨42⟩      Φ := get_all_nonrecursive(L)
⟨43⟩      Ψ := get_all_nonrecursive(R)
⟨44⟩      θ' := bind_remaining_existential_variables(Ψ, Φ, θ)
          /* Extend θ to θ' by trying obvious bindings
             for remaining existential variables */
⟨45⟩      if (has_existential_variables(Ψ, Φ, θ')) continue
⟨46⟩      if (entailment(Φ, Ψθ')) return θ'
⟨47⟩  return ⊥
endfunction
```

Figure 5.13: Supporting Functions

Recursive Call: When the attempt of direct proof is not successful, we collect all possible transformations of the current proof obligation, using (IA-1), (IA-2), (LU+I), (RU) rules, into *a set of set of* obligations $OrSet$. The current proof obligation can be successfully discharged if there is *any* set of proof obligations $Obs \in OrSet$, where we can discharge *every* proof obligation $ob \in Obs$. The realization of the proof rules in our algorithm is straightforward, except for a few noteworthy points:

1. Our induction applications will not exhaustively search for all possible candidates. Instead, we only search for some trivial renaming which meets the side conditions of the rules.

2. When we perform left unfold, an obligation which is trivially true (trivially_true), i.e. the non-recursive part of the LHS is unsatisfiable, is immediately removed.

3. If the current obligation contains the LHS and RHS which contradict each other (contradict), right unfold will be avoided. The proof for this obligation can succeed only if there are no models for the LHS (so only left unfolds are required).

We note that our proof search proceeds recursively in a depth first search manner. The order in which the sets of obligations $Obs \in OrSet$ are considered might heavily affect the

efficiency, i.e. the running time, but *not* the *effectiveness*, i.e. the ability to prove, of our framework. Such order is dictated by our heuristics, as the call to function OrderByHeuristics (line 30) indicates. We remark that our heuristics, described below, are very *intuitive* and directly follow from the fact that our base case is reached by a successful direct proof.

We proceed by a number of passes. In each pass, we first order the obligations within each $Obs \in OrderSet$. We then consider the order of $OrderSet$ by comparing the last obligation in each set $Obs \in OrderSet$. Subsequent passes will not undo the work of the previous passes, but instead work on the obligations and/or sets of obligations which are tied in previous passes.

1. An obligation which has contradicting LHS and RHS, given by the function contradict will be ordered after those do not (since the chance to successfully discharge such obligation is small).

2. An obligation contains no recursive predicates on the RHS will be order before those contain some recursive predicate(s) on the RHS.

3. An obligation having a recursive predicate q such that q appears in the RHS but not in the LHS will be ordered after those not.

4. An obligation contains more existential variables which cannot be deterministically bound to some non-existential variables (variables on the LHS) will be ordered after those contains less.

5. An obligation resulted from a left unfold will be ordered before those resulted from a right unfold (since it allows an induction hypothesis to be added).

Example 7. *Revisit the obligation in Example 4, but now with the starting set of assumptions to be empty:*

$$\emptyset \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2 \,.$$

For simplicity we ignore the information about the counters lb, rb, and ib. First, the call to DirectProof fails since there is the predicate ls which appears in the RHS but not in the LHS. Induction rules cannot take place either, as the set of assumptions is currently empty. We proceed by performing a left unfold first. Note that there is only one recursive predicate on the LHS. Let $\tilde{A}$ be:

$$\{\langle \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2; \mathsf{list}(x, L) \rangle\}.$$

The result for our left unfold is a set of two obligations:

$$O_0 \equiv \{\tilde{A}' \vdash x = 0, L \doteq \Omega \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2\,;$$
$$\tilde{A} \vdash L \doteq (x \mapsto t) * L', \mathsf{list}(t, L') \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2\}$$

We proceed with right unfold, producing four sets, each consists of one (simplified) obligation as follows:

$$O_1 \equiv \{\emptyset \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), y = 0, L_2 \doteq \Omega, L \doteq L_1 * L_2\}$$
$$O_2 \equiv \{\emptyset \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(t, L_3), L \doteq L_1 * (y \mapsto t) * L_3\}$$
$$O_3 \equiv \{\emptyset \vdash \mathsf{list}(x, L) \models x = y, \mathsf{list}(y, L_2), L \doteq \Omega * L_2\}$$
$$O_4 \equiv \{\emptyset \vdash \mathsf{list}(x, L) \models x \neq y, \mathsf{ls}(x, t, L_3), \mathsf{list}(y, L_2), L \doteq (t \mapsto y) * L_3 * L_2\}$$

Assume that the initial order of those sets of obligations are as shown above. After the first two passes, the order between those sets is the same. The third pass, however, moves the singleton set O_3 to the first position. The fourth pass, on the other hand, moves O_1 to the second position. The fifth pass keep O_0 at the third position. The remaining two singleton sets, namely O_2 and O_4 are tied and placed at the end.

We proceed with the first obligation set, namely O_3, and a direct proof of it is successful. Therefore the original obligation can be discharged. The corresponding sequence of the proof rules is shown below, which is slightly different from what shown in Fig. 5.7.

$$\dfrac{\dfrac{\dfrac{}{\texttt{True}}\,(\textsc{cp})}{\emptyset \vdash \mathsf{list}(x, L) \models x = x, L \doteq \Omega * L}\,(\textsc{sub})}{\emptyset \vdash \mathsf{list}(x, L) \models x = y, \mathsf{list}(y, L_2), L \doteq \Omega * L_2}\,(\textsc{ru})$$
$$\emptyset \vdash \mathsf{list}(x, L) \models \mathsf{ls}(x, y, L_1), \mathsf{list}(y, L_2), L \doteq L_1 * L_2$$

5.6 Soundness

Theorem 1 (Soundness)**.** *An entailment $\mathcal{L} \models \mathcal{R}$ holds if, starting with $\emptyset \vdash \mathcal{L} \models \mathcal{R}$, there exists a sequence of applications of proof rules that results in an empty set of proof obligations.*

Proof Sketch. The soundness of rule (CP) is obvious. The rule (RU) is sound because when $\mathcal{R}' \in \textsc{unfold}_q(\mathcal{R})$ then $\mathcal{R}' \models \mathcal{R}$. Therefore, the proof of $\tilde{A} \vdash \mathcal{L} \models \mathcal{R}$ can be replaced by the proof of $\tilde{A} \vdash \mathcal{L} \models \mathcal{R}'$ since $\mathcal{L} \models \mathcal{R}'$ is stronger than $\mathcal{L} \models \mathcal{R}$. Similarly, the rule (SUB) is sound because $\mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R}\theta$ and $\mathcal{L} \wedge p(\tilde{x}) \models_{\mathtt{CP}} \tilde{x} = \tilde{y}\theta$ is stronger than the $\mathcal{L} \wedge p(\tilde{x}) \models \mathcal{R} \wedge p(\tilde{y})$.

The rule (LU+I) is *partially sound* in the sense that when $\textsc{unfold}_p(\mathcal{L}) = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, then proving $\mathcal{L} \models \mathcal{R}$ can be substituted by proving $\mathcal{L}_1 \models \mathcal{R}, \ldots, \mathcal{L}_n \models \mathcal{R}$. This is because in the least-model semantics of the definitions, $\mathcal{L}$ is equivalent to $\mathcal{L}_1 \vee \ldots \vee \mathcal{L}_n$. However,

whether the addition of $\langle \mathcal{L} \models \mathcal{R}; p \rangle$ to the set of assumed obligations $\tilde{A}$ is sound depends on the use of them in the application of (IA-1) and (IA-2).

We now proceed to prove the soundness of (IA-1) and (IA-2). First, define a *refutation* to an obligation $\mathcal{L} \models \mathcal{R}$ as a successful derivation of one or more atoms in $\mathcal{L}$ whose answer Ψ has an instance (ground substitution) β such that $\Psi\beta \wedge \mathcal{R}\beta$ is false. A finite refutation corresponds to a such derivation of finite length. A nonexistence of finite refutation means that $lm(P) \models (\mathcal{L} \rightarrow \mathcal{R})$, or in other words, $\mathcal{L} \models \mathcal{R}$. A derivation of a refutation is obtainable by left unfold (LU+I)) rule only. Hence a finite refutation of length k implies a corresponding k left unfold (LU+I) applications that results in a contradiction.

In the rules (IA-1) and (IA-2), we assume the hypothesis $A\theta$, where $A \equiv \mathcal{L}' \models \mathcal{R}'$ is some entailment encountered previously. By having the side conditions proved separately, we then can soundly transform the current entailment B into a new entailment C. In case of (IA-1), $B \equiv p(\tilde{x}) \wedge \mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}$ and $C \equiv \mathcal{R}'\theta \wedge \mathcal{L}_2 \models \mathcal{R}$. In case of (IA-2), we have $B \equiv p(\tilde{x}) \wedge \mathcal{L}_1 \models \mathcal{R}$ and $C \equiv \mathcal{L}_1 \models \mathcal{L}'\theta$.

Notice that the side conditions ensure that $A\theta \implies (C \implies B)$, where $\implies$ denotes implication. The side conditions also enforce the renaming θ to "progress" at least the left unfold of recursive atom $p(\tilde{y})$ to match with a newly generated atom $p(\tilde{x})$. This indeed enforces a well-founded measure on A.

To be more concrete, note that our transformation from B to C is *unsound only if* there exists a refutation β to B, and therefore A, but β is not a refutation to C. We then proceed to prove by contradiction. W.l.o.g., assume β is such a refutation and is the refutation to A which has the smallest length k. Trivially $k > 0$ as A has no finite refutation of length 0. Since there is at least one left unfold from A to B, β must be a refutation to B but of length less than equal to k. However, since $A\theta \implies (C \implies B)$, and β is a refutation of B but not C, therefore β is also a refutation of $A\theta$. Since θ must "progress" A by at least one left unfold, we end up with the fact that A has a refutation of length less than k. This is a contradiction. □

5.7 Experiments

In our modular verification framework (with the frame rule), the problem of verifying big programs reduces to proving the kinds of verification conditions addressed in this thesis. Our experiments are thus focused on the complexity of the program properties to be proven

instead of the size of programs.

Our evaluations are performed on a 3.2GHz Intel processor with 2GB RAM, running Linux. We evaluated our prototype on a comprehensive set of benchmarks, including both academic algorithms and real programs. The benchmarks are collected from existing systems [Nguyen and Chin, 2008; Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013; Brotherston *et al.*, 2012], those considered as the state-of-the-art for the purpose of proving user-defined recursive data-structure properties in imperative languages. Some of them are also used in the competition SMT-COMP 2014 (Separation Logic)[3]. Note that, in this competition (where lemmas are discouraged), the benchmarks are of the same scale as ours, though ours contain more benchmarks having shape and data properties intertwined, making previous techniques fail to prove. We first demonstrate our evaluation with benchmarks that the state-of-the-art can handle, then with ones that are beyond their current supports.

5.7.1 Within the State-of-the-art

In this subsection, we consider the set of proof obligations where the state-of-the-art, e.g., U+M and "Cyclic Proof", are effective. The purpose of this study is to evaluate the *efficiency* of our implementation against existing systems. This exercise serves as a sanity check for our implementation.

We first start with proof obligations where U+M can automatically discharge without the help of user-defined lemmas. They are collected from the benchmarks of U+M frameworks [Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013]. As expected, our prototype proves all of those obligations; the running time for each is negligible (~ 0.2 second). This is because the proof obligations usually require just either *one* left unfold or *one* right unfold before matching (a direct proof) can successfully take place.

The second set of benchmarks are from "Cyclic Proof" [Brotherston *et al.*, 2012], which are also used in SMT-COMP 2014 (Separation Logic). They are proof obligations which involve unmatchable predicates, thus U+M will not be effective. We also succeed in proving all of those obligations, less than a second for each.

In summary, the results demonstrate that (1) our prototype is able to handle what the state-of-the-art can; (2) our implementation is *competitive* enough.

[3]See https://github.com/mihasighi/smtcomp14-sl

5.7.2 Beyond the State-of-the-art

We now demonstrate the *key* result of this work: proving what are beyond the state-of-the-art.

Proving User-Defined Lemmas: Our prototype can prove all commonly used lemmas, collected from [Nguyen and Chin, 2008; Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013], which U+M and "Cyclic Proof" cannot handle. The running time is always less than a second for each lemma. Table 5.1 shows a non-exhaustive list of common user-defined lemmas. We purposely abstract them from the original usage in order to make them general and representative enough. The lemmas are written in traditional Separation Logic syntax for succinctness. The aim is to give the readers the intuitive meaning of those lemmas though the actual definitions of the predicates must be written in our assertion language, where each predicate will be accompanied by an explicit heap as in our presented examples. Note that due to the duality of the definitions for list segments, e.g., ls vs. $\widehat{\mathsf{ls}}$, each lemma containing them would usually has a dual version, which for space reason we do not list down in Table 5.1. Similarly, some extensions, e.g., to capture the relationship of collective data values (using sets or sequences) between the LHS and the RHS, while can be automatically discharged by our prototype, are not listed in the table.

Table 5.1: Proving Lemmas (existing systems cannot prove).

Lemma
$\mathsf{sorted\_list}(x, min) \models \mathsf{list}(x)$
$\mathsf{sorted\_list}_1(x, len, min) \models \mathsf{list}_1(x, len)$
$\mathsf{sorted\_list}_1(x, len, min) \models \mathsf{sorted\_list}(x, min)$
$\mathsf{sorted\_ls}(x, y, min, max) * \mathsf{sorted\_list}(y, min_2)$ $\wedge\ max \leq min_2 \models \mathsf{sorted\_list}(x, min)$
$\mathsf{ls}(x, y) * \mathsf{list}(y) \models \mathsf{list}(x)$
$\mathsf{ls}(x, y) \models \widehat{\mathsf{ls}}(x, y)$ and $\widehat{\mathsf{ls}}(x, y) \models \mathsf{ls}(x, y)$
$\widehat{\mathsf{ls}}_1(x, y, len_1) * \widehat{\mathsf{ls}}_1(y, z, len_2) \models \widehat{\mathsf{ls}}_1(x, z, len_1{+}len_2)$
$\mathsf{ls}_1(x, y, len_1) * \mathsf{list}_1(y, len_2) \models \mathsf{list}_1(x, len_1{+}len_2)$
$\widehat{\mathsf{ls}}_1(x, last, len) * (last \mapsto new) \models \widehat{\mathsf{ls}}_1(x, new, len + 1)$
$\mathsf{dls}(x, y) * \mathsf{dlist}(y) \models \mathsf{dlist}(x)$
$\widehat{\mathsf{dls}}_1(x, y, len_1) * \widehat{\mathsf{dls}}_1(y, z, len_2) \models \widehat{\mathsf{dls}}_1(x, z, len_1{+}len_2)$
$\mathsf{dls}_1(x, y, len_1) * \mathsf{dlist}_1(y, len_2) \models \mathsf{dlist}_1(x, len_1{+}len_2)$
$\mathsf{avl}(x, hgt, min, max, balance) \models \mathsf{bstree}(x, hgt, min, max)$
$\mathsf{bstree}(x, height, min, max) \models \mathsf{bintree}(x, height)$

Let us briefly comment on Table 5.1. The first group talks about sorted linked lists. As an

example, the second lemma is to state that a sorted list with length *len* and the minimum element *min* is also a list with the same length. The second, third and fourth groups are related to singly-linked lists, doubly-linked lists, and trees respectively.

Verifying Programs without Using Lemmas: Lemmas can serve many purposes. One of its important usage in U+M systems is to equip a proof system with the power of user-provided re-writing rules, to overcome the main limitation of unfold-and-match. However, in the context of program verification, eliminating the usage of lemmas is crucial for improving the performance, because lemma applications, coupled with unfolding, often induce large search space.

We now use a subset of academic algorithms and open-source library programs[4], collected and published by [Chin *et al.*, 2012; Qiu *et al.*, 2013], to demonstrate that our prototype can verify these programs without even stating the appropriate lemmas. The library programs include Glib open source library, the OpenBSD library, the Linux kernel, the memory regions and the page cache implementations from two different operating systems. While Table 5.2 summarizes the verification of data structures from academic algorithms, Table 5.3 reports on open-source library programs.

Table 5.2: Verification of Academic Algorithms (existing systems require lemmas).

DS	Function	T/F
Sorted List	`find_last_iter, insert_iter, quick_sort_iter, bubble_sort`	$<1s$
Circular List	`count`	$<1s$
BST	`insert_iter,find_leftmost_iter remove_root_iter, delete_iter`	$<1s$

Remark #1: Using automatic induction, we have successfully eliminated the requirement for lemmas in existing systems (e.g., [Chin *et al.*, 2012; Qiu *et al.*, 2013]) for proving the functional correctness of the programs in Table 5.2 and 5.3. As already stated in Section 5.1, existing systems require lemmas in two common scenarios. First, it is when the traversal order of the data structures is different from what suggested by the recursive definitions, e.g., `OpenBSD/queue.h`. Second, it is due to the boundaries caused by iterative loops or multiple function calls. One example is `append` function in `glib/gslist.c`, where (in addition to the list definition) the list segment, ls(head,last), is necessary to say about the function invariant

[4]See `http://www.cs.uiuc.edu/~madhu/dryad/sl`

— the last node of a non-empty input list is always reachable from the list's head. Other examples are to make a connection between a sorted list and a singly-linked list (e.g., in sorting algorithms), between two sorted partitions (e.g. in `quick_sort_iter`), between a circular list and a list segment (e.g., `count`), etc.

Table 5.3: Verification of Open-Source Libraries (existing systems require lemmas).

Program	Function	T/F
glib/gslist.c Singly Linked-List	`find, position, index, nth,last,length,append, insert_at_pos,merge_sort, remove,insert_sorted_list`	$<1s$
glib/glist.c Doubly Linked-List	`nth, position, find, index, last, length`	$<1s$
OpenBSD/ queue.h Queue	`simpleq_remove_after, simpleq_insert_tail, simpleq_insert_after`	$<1s$
ExpressOS/ cachePage.c	`lookup_prev, add_cachepage`	$<1s$
linux/mmap.c	`insert_vm_struct`	$<1s$

Remark #2: The verification time for each function is always less than 1 second. This is within our expectation because whenever our proof method succeeds, the size of the proof tree is relatively small. For example, in order to prove the functional correctness of `append` function in `glib/gslist.c`, we only need to prove 3 obligations, each of which requires no more than two left unfolds, two right unfolds and two inductions[5]. In fact, the maximum number of left unfolds, right unfolds and inductions used in our system are 5, 5 and 3 respectively, even for the functions that take U+M frameworks much longer time to prove. For example, consider `simpleq_insert_after`, a function to insert an element into a queue. This example requires reasoning about unmatchable predicates: to prove it DRYAD needs 18 seconds and the help from a lemma. Such inefficiency is due to the use of a complicated lemma[6], which consists of a large disjunction. Though efficient in practice, SMT solvers still face a combinatorial explosion challenge as they dissect the disjunction. In other words, in addition to having a higher level of automation, our framework has a potential advantage

[5]Since the number of rules (disjuncts) in a predicate definition is fixed and usually small, the size of proof tree mainly depends on the number of unfolds and inductions.

[6]We believe that the lemmas in [Qiu *et al.*, 2013] are unnecessarily complicated, because the authors want to reduce the number of them, by grouping a few into one.

of being more efficient than existing U+M systems.

5.8 Related Work

There is a vast literature on program verification considering data structures. The well known formalism of Separation Logic (SL) [Reynolds, 2002a] is often combined with a recursive formulation of data structure properties. Implementations, however, are incomplete, e.g., [Berdine *et al.*, 2005; Iosif and abd J. Simachek, 2013], or deal only with fragments [Berdine *et al.*, 2004; Magill *et al.*, 2008]. There is also literature on decision procedures for restricted heap logics; we mention just a few examples: [Rakamaric *et al.*, 2007a; Rakamaric *et al.*, 2007b; Lahiri and Qadeer, 2008; Ranise and Zarba, 2006; Bouajjani *et al.*, 2009a; Bjørner and Hendrix, 2009]. These have, however, severe restrictions on expressivity. None of them can handle the VC's of the kind considered in this thesis.

There is also a variety of verification tools based on classical logics and SMT solvers. Some examples are Dafny [Leino, 2010], VCC [Cohen *et al.*, 2009] and Verifast [Jacobs *et al.*, 2011] which require significant ghost annotations, and annotations that explicitly express and manipulate frames. They do not automatically verify the general and complex obligations addressed in this thesis; but such obligations are often resorted to interactive theorem provers, e.g., Mona, Isabelle or Coq, enabling manual guidance from the users.

Navarro and Rybalchenko showed that significant performance improvements can be obtained by incorporating first-order theorem proving techniques into SL provers [Navarro and Rybalchenko, 2011]. However, the focus of that work is about list segments, not general user-defined recursive predicates. On a similar thread, [Piskac *et al.*, 2013] advances the automation of SL, using SMT, in verifying procedures manipulating list-like data structures. The works [Zee *et al.*, 2008; Zee *et al.*, 2009; Chin *et al.*, 2012; Madhusudan *et al.*, 2012; Qiu *et al.*, 2013] are also closely related: they form the U+M paradigm which we have carefully discussed in Section 5.1 and 5.2.

In the literature, there have been works on automatic induction [Boyer and Moore, 1990; Dillinger *et al.*, 2007; Leino, 2012; Sonnex *et al.*, 2012]. They are concerned with proving a *fixed* hypothesis, say $h(\tilde{x})$, that is, to show that $h()$ holds over all values of the variables $\tilde{x}$. The challenge is to discover and prove $h(\tilde{x}) \implies h(\tilde{x}')$, where expression $\tilde{x}$ is less than the expression $\tilde{x}'$ in some well-founded measure. Furthermore, a *base case* $h(\tilde{x}_0)$ needs to be proven. Automating this form of induction usually relies on the fact that some subset

of $\tilde{x}$ are variables of *inductive types*. In contrast, our notion of induction hypothesis is completely different. First, we do not require that some variables are of inductive (and well-founded) types. Second, the induction hypotheses are not supplied explicitly. Instead, they are constructed implicitly via the discovery of a valid proof path. This allows much more potential for automating the proof search. Third (and this also applied to the "Cyclic Proof" method mentioned below), multiple induction hypotheses can be exploited within a single proof path. Without this, as a concrete example, we would not be able to prove $\widehat{\mathsf{ls}}(x,y) \models \mathsf{ls}(x,y)$.

We further highlight the work of Lahiri and Qadeer [Lahiri and Qadeer, 2006], which adapts the induction principle for proving properties of well-founded linked list. The technique relies on the well-foundedness of the heap, while employing the induction principle to derive from two basic axioms a small set of additional first-order axioms that are useful for proving the correctness of several simple programs.

We now mention works on "Cyclic Proof", e.g., [Brotherston *et al.*, 2011; Brotherston *et al.*, 2012]; and also a somewhat related concept called "Matching Logic" [Rosu and Stefanescu, 2012]. "Cyclic Proof" replaces explicit induction reasoning by detecting well-founded *infinite descent* over the cyclic proof graphs. (We note that the current implementations of "Cyclic Proof" [Brotherston *et al.*, 2011; Brotherston *et al.*, 2012], however, are very limited.) The crucial departure from our work in this thesis is that the above-mentioned methods do not deal with the notion of *applying an induction step* in order to *generate* a new and different proof obligation. The power of our methodology comes from the fact that the induction step can be applied repetitively along a proof path, as in the proof of $\widehat{\mathsf{ls}}(x,y) \models \mathsf{ls}(x,y)$.

We finally mention the work [Jaffar *et al.*, 2008], from which the concept of our automatic induction originates. The current work extends [Jaffar *et al.*, 2008] first by refining the original single coinduction rule into two more powerful rules, to deal with the antecedent and consequent of a VC respectively. Secondly, the *application* of the rules has been systematized so as to produce a rigorous proof search strategy. Another technical advance is our introduction of *timestamps* (a progressive measure) in the two induction rules as an efficient technique to avoid circular reasoning. Finally, the present work focuses on program verification and uses a specific domain of discourse involving the use of explicit symbolic heaps and separation.

5.9 Concluding Remarks

We presented a framework for proving recursive properties of data structures providing a new level of automation across a wider class of programs. Its key technical feature is the automatic use of induction. More specifically, the framework allows for selecting a dynamically generated proof obligation as an induction hypothesis, and then using this formula in an induction step in order to generate a new proof obligation. The main technical challenge of avoiding circular reasoning was overcome by an intricate restriction on variable renamings. Finally, experimental evidence was presented to show that many real-life proofs, including those of lemmas whose unproved use has been necessary in previous systems, can now be fully automated.

Chapter 6

Conclusion and Future Work

In this chapter, we first summarize the thesis contributions and then discuss their foreseeable impacts and future works.

6.1 Summary

In this thesis, we have proposed three systematic techniques to reason about unbounded data structures. In here, we briefly summarize these main contributions.

The first technique is to implement *lazy reasoning* methodology. Its introduction is to mitigate the problem of combinatorial explosion in searching for a solution of the input constraints. We have applied this technique in building an *efficient* string solver. Specifically, we incrementally reduce recursive predicates, which are used to represent string operations, via splitting (and/or unfolding) process, until their subparts are bounded with constant strings/characters to be consumed. While modern string solvers exist, they suffer in one way or another: (1) the constraint language may not be expressive enough (even though the solver is fast); or (2) the solver may not be fast enough to accommodate realistically large programs. Thanks to lazy reasoning, we now have a fast symbolic string solver to support an expressive language. Experimental evaluations show that our string solver S3, despite being more expressive than other solvers, is much more robust and efficient. In practice, S3 is recently used as a back-end in program analyzers such as [Xie *et al.*, 2015; Xiaofei, 2016].

Since lazy reasoning does not address non-termination issues, we have next proposed two novel methods: progressive reasoning and inductive reasoning. *Progressive reasoning* aims to address non-termination in solving string constraints. The key feature of our algorithm

is a pruning method on the subproblems, in a way that is *directed*. More specifically, our algorithm detects non-progressive scenarios with respect to a criterion of minimizing the "lexicographical length" of the returned solution, if a solution in fact exists. Informally, in the search process based on reduction rules, we can soundly prune a subproblem when the answer we seek can be found more efficiently elsewhere. Experimental evaluations show the promising results of our new string solver S3P in dealing with non-termination in string solving. Furthermore, because our algorithm deals with recursive definitions in a somewhat general manner, we believe it can be extended to support reasoning about other unbounded data structures, for example heap-allocated data structures.

To facilitate the need from security analyses of web applications, we have also made two other technical contributions in order to improve the solver's performance. The first improvement is the bi-directional interaction between the string solver and the integer solver of Z3. This allows the string solver not only to propagate its length information to integer solver, but also to query about the relationship between the lengths of string variables from the integer solver. The information ultimately gives us a truly incremental solver for both string and non-string constraints. The second improvement is to support conflict clause learning for the string solver. Here, we want to produce a set of *conflict clauses*, a generalization of the input formula, that is now known to be unsatisfiable. The key technical challenge is, how conflict clause learning can work in tandem with the pruning of non-progressive formulas, because at the time of pruning, the unsatisfiability of the input formula is unknown. These two improvements have been demonstrated to show usefulness in pruning the search space and new levels of results in JavaScript benchmarks arising from web applications.

Finally, we have proposed a general method that includes *inductive reasoning* for entailment proving. It aims to address non-termination in proving dynamically-allocated data structure properties. The challenge is how to use induction correctly and avoid erroneous proof arising from a form of *circular reasoning*. Our method is able to use *dynamically generated* formulas as induction hypotheses, and to enforce an *anti-circular* condition so that any application of an induction step is guaranteed to be correct. The state-of-the-art methods are often unable to prove relationship between different data structures (e.g. to prove that a sorted list is a list). As a result, they would not be able to automatically verify a large class of programs. Inductive reasoning helps us to close such remaining gap in ex-

isting systems. More importantly, it also gets us back the power of compositional reasoning in dealing with user-defined recursive predicates that are used to represent data structures properties.

6.2 Future Work

We first mention a few important applications of string solving for web security. The most important one is to apply string solving in web security analysis. Although we have seen significant advances in the general area of software reasoning (e.g. [McMillan, 2010; Beyer, 2013]), some fundamental breakthroughs, especially in constraint solving and program analysis, are still needed to enhance the security of web applications.

It is generally accepted that the holy grail of a static analyzer which can accurately pinpoint vulnerabilities is not achievable. Instead, the general methodologies of concolic testing [Godefroid *et al.*, 2005] and dynamic symbolic execution (DSE), e.g. [Schwartz *et al.*, 2010], have been shown to be successful in the sense that they can detect significant cases of vulnerabilities, and yet have a good coverage of the space of all possible program execution paths. The successful applications, however, have so far been largely limited to program testing (e.g., Kudzu [Saxena *et al.*, 2010], Jalangi [Sen *et al.*, 2013], SymJS [Li *et al.*, 2014]). That is, once given a program path, the vulnerability issue is settled by deciding if the associated logical formula to the path is consistent or not. In such case, solving and finding a model for a path constraint formula play an important role in determining real security attacks.

However, there is little work on program analysis. That is, to scan a significant (but not a total) portion of the space of program paths, and to discover, not just test, some important properties of these paths. As a concrete example, taint analysis [Newsome, 2005; Tripp *et al.*, 2009; Arzt *et al.*, 2014; Cai *et al.*, 2016] often helps highlight specific security risks primarily associated with web sites which are attacked using techniques such as SQL injection or buffer overflow attacks. Therefore, the desired is a broad framework of dynamic symbolic execution to support not just testing but also other typical analyses, closely related to security research such as taint analysis, information flow (leakage) analysis. Extending from testing to these forms of analysis is a significant contribution. Current optimization techniques in program testing/verification are not very applicable in this context. In addition, it is foreseeable that a more powerful, robust, and efficient string solver would be a major component for this

new framework.

Another application of string solving is model counting. In fact, the model counting problem, which is to compute the number of satisfying assignments of a set of input constraints, already arises in many fields of computer science including artificial intelligence, program optimizations. For example, it is used in probabilistic inference problems in Bayesian networks [Bacchus *et al.*, 2009; Bayardo and Pehoushek, 2000; Roth, 1996], in memory size minimization [Turjan *et al.*, 2002], worst case execution time estimation [Kirner *et al.*, 2002], increasing parallelism [Turjan *et al.*, 2002], and improving cache effectiveness [Beyls and D'Hollander, 2005].

In particular, model counting also has security applications. Specifically, model counters can be used directly by quantitative analyses of information flow (in order to determine how much secret information is leaked), program execution time, combinatorial circuit designs, and probabilistic reasoning. For example, the constraints can be used to represent the relation between the inputs and outputs implied by the program in quantitative theories of information flow. This in turn has numerous applications such as quantitative information flow analysis [Smith, 2009; Backes *et al.*, 2009; Eldib *et al.*, 2014; Bang *et al.*, 2016], differential privacy [Alvim *et al.*, 2011], secure information flow [Sabelfeld and Myers, 2006], anonymity protocols [Chatzikokolakis *et al.*, 2008], and side-channel analysis [Köpf and Basin, 2007]. Recently, model counting is also used by probabilistic symbolic execution where the goal is to compute probability of the success and failure program paths [Filieri *et al.*, 2013; Borges *et al.*, 2014].

Given that strings are ubiquitous in web applications, the model counting problem for the string domain is, therefore, even of more interest. However, though there are a lot of works on model counting for different kinds of domains such as boolean [Biondi *et al.*, 2013], and integer domains [Morgado *et al.*, 2006], there is little work for the string domain. The reason is that most of existing techniques are only applicable to "bounded" domains (e.g., bit vector is a fixed-length array of bits). By contrast, string is an *unbounded* data structure. For example, though we can still represent a bounded string as a bit vector and then employ the existing model counting for bit vector constraints to calculate the number of models, this approach may not scale to complex string constraints. Specifically, according to [Kiezun *et al.*, 2009a; Saxena *et al.*, 2010], the constraints representing the regular expression `S.`**`match`**`/(a | b)*/` as bit vectors can grow exponentially in the size of

the input.

On the other hand, existing model counting technique for the string domain (e.g. [Luu *et al.*, 2014], [Aydin *et al.*, 2015]) are not precise enough, especially when the input constraints also include string lengths. For example, the technique in [Aydin *et al.*, 2015], which represents all solutions of the input constraints as an automata before counting the accepting paths of the constraint DFA up to a given length bound, can only count precisely when the solution set is captured by using an automaton. This technique thus no more counts precisely if the solution set is beyond a regular language. We believe the recent improvement in string solving can help us to achieve a more systematic model counting method for the string domain.

Finally, we mention applications of our inductive reasoning in automated verification of very large heap-manipulating programs. The advantage of program verification is obviously well-known. That is to guarantee absence of certain classes of errors such as memory safety errors. This is especially useful for applications where correctness is particularly important such as car braking systems, medical equipment, voting machines [Sastry *et al.*, 2006; Sturton *et al.*, 2009; Srivastava and Schumann, 2013]. In practice, automated verification has been already applied in large code bases such as seL4 kernel implementation [Klein *et al.*, 2009]. However, a lot of lemmas are still used to prove the program correctness. We believe by using our proof technique many of them can be eliminated, which in turn helps to improve the performance of the whole verification process.

Bibliography

[Abdulla *et al.*, 2014] P. Abdulla, M. Atig, Y.-F. Chen, L. Holk, A. Rezine, P. Rmmer, and J. Stenman. String constraints for verification. In *CAV*, pages 150–166. Springer, 2014.

[Abdulla *et al.*, 2015] P. Abdulla, M. Atig, Y.-F. Chen, L. Holk, A. Rezine, P. Rmmer, and J. Stenman. Norn: An smt solver for string constraints. In *CAV 2015*, pages 462–469. Springer, 2015.

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Alkhalaf *et al.*, 2012a] Muath Alkhalaf, Tevfik Bultan, and Jose L. Gallegos. Verifying client-side input validation functions using string analysis. In *ICSE*, pages 947–957, 2012.

[Alkhalaf *et al.*, 2012b] Muath Alkhalaf, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso, and Christopher Kruegel. Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. In *ISSTA*, pages 56–66. ACM, 2012.

[Alvim *et al.*, 2011] Mario S. Alvim, Miguel E. Andres, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative information flow and applications to differential privacy. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design VI: FOSAD Tutorial Lectures*, pages 211–230, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Arzt *et al.*, 2014] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[Avgerinos *et al.*, 2014] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, pages 1083–1094. ACM, 2014.

[Axelsson *et al.*, 2008] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *ICALP*, pages 410–422. Springer-Verlag, 2008.

[Aydin *et al.*, 2015] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *CAV 2015*, pages 255–272, 2015.

[Bacchus *et al.*, 2009] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Solving #sat and bayesian inference with backtracking search. *J. Artif. Int. Res.*, 34(1):391–442, March 2009.

[Backes *et al.*, 2009] M. Backes, B. Kpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, May 2009.

[Bang *et al.*, 2016] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 193–204, New York, NY, USA, 2016. ACM.

[Barcelo *et al.*, 2012] Pablo Barcelo, Diego Figueira, and Leonid Libkin. Graph logics with rational relations and the generalized intersection problem. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 115–124, Washington, DC, USA, 2012. IEEE Computer Society.

[Barrett *et al.*, 2009] 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 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[Barrett *et al.*, 2016] Clark Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. Efficient solving of string constraints for security analysis. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, HotSos '16, pages 4–6, New York, NY, USA, 2016. ACM.

[Bayardo and Pehoushek, 2000] Roberto J. Bayardo, Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 157–162. AAAI Press, 2000.

[Berdine *et al.*, 2004] J. Berdine, C. Calcagno, and P. OHearn. A decidable fragment of separation logic. In *FSTTCS, LNCS 3328*, pages 97–109. Springer, 2004.

[Berdine *et al.*, 2005] J. Berdine, C. Calcagno, and P. OHearn. Symbolic execution with separation logic. In *APLAS, LNCS 3780*, pages 52–68. Springer, 2005.

[Beyer, 2013] Dirk Beyer. Second competition on software verification. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 594–609, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Beyls and D'Hollander, 2005] Kristof Beyls and Erik H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, April 2005.

[Biondi *et al.*, 2013] Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wkasowski. Quail: A quantitative security analyzer for imperative code. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification: 25th International*

Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, pages 702–707, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Bisht *et al.*, 2010] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *CCS*, pages 607–618. ACM, 2010.

[Bisht *et al.*, 2011] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, pages 575–586, 2011.

[Bjørner and Hendrix, 2009] N. Bjørner and J. Hendrix. Linear functional fixed-points. In *CAV, LNCS 5643*, pages 124–139. Springer, 2009.

[Bjørner *et al.*, 2009] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321. Springer-Verlag, 2009.

[Borges *et al.*, 2014] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.

[Bouajjani *et al.*, 2009a] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR, LNCS 5710*, pages 178–195. Springer, 2009.

[Bouajjani *et al.*, 2009b] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. A logic-based framework for reasoning about composite data structures. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory: 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, pages 178–195, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Boyer and Moore, 1990] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. In *CADE*, 1990.

[Brotherston and Villard, 2014] J. Brotherston and J. Villard. Parametric completeness for separation theories. In *POPL*, pages 453–464, 2014.

[Brotherston *et al.*, 2011] J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE*, 2011.

[Brotherston *et al.*, 2012] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *APLAS*, pages 350–367, 2012.

[Brotherston *et al.*, 2016] James Brotherston, Nikos Gorogiannis, Max Kanovich, and Reuben Rowe. Model checking for symbolic-heap separation logic with inductive predicates. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 84–96, New York, NY, USA, 2016. ACM.

[Brumley *et al.*, 2007] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium*, pages 15:1–15:16, 2007.

[Buchi and Senger, 1988] J. R. Buchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, pages 337–342, 1988.

[Bucur *et al.*, 2014] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS*, pages 239–254, 2014.

[Cai *et al.*, 2016] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. Sworddta: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences*, 21(1):10–20, 2016.

[Charatonik and Pacholski, 1991] W. Charatonik and L. Pacholski. Word equations with two variables. In *IWWERT*, volume 677, pages 43–57. Springer, 1991.

[Chatzikokolakis *et al.*, 2008] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Inf. Comput.*, 206(2-4):378–401, February 2008.

[Chaudhuri and Foster, 2010] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *CCS*, pages 585–594, 2010.

[Chin *et al.*, 2012] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In *Science of Computer Programming, 77(9)*, pages 1006–1036, 2012.

[Christensen *et al.*, 2003] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.

[Chu *et al.*, 2015] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 457–466, New York, NY, USA, 2015. ACM.

[Cohen *et al.*, 2009] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs, LNCS 5674*, pages 23–42. Springer, 2009.

[Cormen *et al.*, 2001] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[Dabrowski and Plandowski, 2002] Robert Dabrowski and Wojtek Plandowski. On word equations in one variable. In Krzysztof Diks and Wojciech Rytter, editors, *Mathematical Foundations of Computer Science 2002: 27th International Symposium, MFCS 2002 Warsaw, Poland, August 26–30, 2002 Proceedings*, pages 212–220, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Dabrowski and Plandowski, 2004] Robert Dabrowski and Wojtek Plandowski. Solving two-variable word equations. In Josep Diaz, Juhani Karhumaki, Arto Lepisto, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 408–419, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Dalci, 2012] Eric Dalci. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html, 2012.

[De Moura and Bjørner, 2008a] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.

[De Moura and Bjørner, 2008b] 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.

[De Moura and Bjørner, 2011] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[Diekert and Robson, 1999] Volker Diekert and John Michael Robson. Quadratic word equations. In Juhani Karhumaki, Hermann Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 314–326, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Diekert *et al.*, 1997] Volker Diekert, Yuri Matiyasevich, and Anca Muscholl. Solving trace equations using lexicographical normal forms. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming: 24th International Colloquium, ICALP '97 Bologna, Italy, July 7–11, 1997 Proceedings*, pages 336–346, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Diekert *et al.*, 1999] Volker Diekert, Yuri Matiyasevich, and Anca Muscholl. Solving word equations modulo partial commutations. *Theor. Comput. Sci.*, 224(1-2):215–235, August 1999.

[Dillinger *et al.*, 2007] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon, and J. Strother Moore. ACL2s: "The ACL2 Sedan". In *ICSE*, 2007.

[Duck *et al.*, 2013] G. Duck, J. Jaffar, and Nicolas Koh. Constraint-based program reasoning with heaps and separation. In *CP, LNCS 8124*, 2013.

[Durnev, 1995] V. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Siberian Mathematical Journal*, 36(5):1067–1080, 1995.

[ECMA-404,] ECMA-404. Javascript object notation. http://www.json.org/.

[Eldib *et al.*, 2014] Hassan Eldib, Chao Wang, and Patrick Schaumont. Smt-based verification of software countermeasures against side-channel attacks. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 62–77, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Emmi *et al.*, 2007] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[Filieri *et al.*, 2013] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[Floyd, 1967] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[Furia, 2010] Carlo A. Furia. What's decidable about sequences? In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis: 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, pages 128–142, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Ganesh *et al.*, 2013] 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.

[Gange *et al.*, 2013] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, pages 277–291, 2013.

[Ghosh *et al.*, 2013] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: An automatic test generation tool for industrial java applications with strings. In *ICSE*, pages 992–1001, 2013.

[Godefroid *et al.*, 2005] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[Godefroid *et al.*, 2008] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[Halfond *et al.*, 2009] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA*, pages 285–296, 2009.

[He *et al.*, 2013] Jun He, Pierre Flener, Justin Pearson, and WeiMing Zhang. Solving string constraints: The case for constraint programming. In *CP*, pages 381–397, 2013.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hooimeijer and Weimer, 2009] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, pages 188–198, 2009.

[Hooimeijer and Weimer, 2010] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *ASE*, pages 377–386. ACM, 2010.

[Ilie and Plandowski, 2000] Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *RAIRO-Theor. Inf. Appl.*, 34(6):467–501, 2000.

[Iosif and abd J. Simachek, 2013] R. Iosif and A. Rogalewicz abd J. Simachek. The tree width of separation logic with recursive definitions. In *CADE, LNAI*. Springer, 2013.

[Ishtiaq and O'Hearn, 2001] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[Itzhaky *et al.*, 2013] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 756–772, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Jacobs *et al.*, 2011] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Nasa Formal Methods Symposium, LNCS 6617*, pages 41–55. Springer, 2011.

[Jaffar and Maher, 1994] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.

[Jaffar *et al.*, 2008] J. Jaffar, A. E. Santosa, and R. Voicu. A coinduction rule for entailment of recursively defined properties. In *CP, LNCS 1454*, pages 493–508, 2008.

[Jaffar *et al.*, 2013] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *FSE*, pages 48–58. ACM, 2013.

[Jensen *et al.*, 2013] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *ISSTA*, pages 67–77, 2013.

[Jeż, 2016] Artur Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, February 2016.

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

[Kiezun *et al.*, 2009a] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116. ACM, 2009.

[Kiezun *et al.*, 2009b] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[Kirner *et al.*, 2002] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully automatic worst-case execution time analysis for matlab/simulink models. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, ECRTS '02, pages 31–, Washington, DC, USA, 2002. IEEE Computer Society.

[Klein *et al.*, 2009] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Köpf and Basin, 2007] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 286–296, New York, NY, USA, 2007. ACM.

[Lahiri and Qadeer, 2006] S. K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126. ACM, 2006.

[Lahiri and Qadeer, 2008] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, pages 171–182. ACM, 2008.

[Leino, 2010] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR, LNCS 6355*, pages 348–370. Springer, 2010.

[Leino, 2012] K. R. M. Leino. Automating induction with an smt solver. In *VMCAI*, 2012.

[Li *et al.*, 2014] Goudong Li, Esben Andreasen, and Ghosh Indradeep. Symjs: Automatic symbolic testing of javascript web applications (to appear). In *FSE*, FSE'14, 2014.

[Liang *et al.*, 2014] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A dpll(t) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662. Springer, 2014.

[Liang *et al.*, 2015] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. A decision procedure for regular membership and length constraints over unbounded strings. In Carsten Lutz and Silvio Ranise, editors, *Proceedings of the 10th International Symposium on Frontiers of Combining Systems (FroCoS '15)*, volume 9322 of *Lecture Notes in Artificial Intelligence*, pages 135–150. Springer, September 2015. Wroclaw, Poland.

[Liang *et al.*, 2016] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Form. Methods Syst. Des.*, 48(3):206–234, June 2016.

[Lin and Barceló, 2016] Anthony W. Lin and Pablo Barceló. String solving with word equations and transducers: Towards a logic for analysing mutation xss. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 123–136, New York, NY, USA, 2016. ACM.

[Luu *et al.*, 2014] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 565–576, New York, NY, USA, 2014. ACM.

[Madhusudan *et al.*, 2012] P. Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 611–622. ACM, 2012.

[Magill *et al.*, 2008] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV, LNCS 5123*, pages 428–432. Springer, 2008.

[Makanin, 1977] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129, 1977.

[Maras *et al.*, 2013] Josip Maras, Maja Štula, and Jan Carlson. Generating feature usage scenarios in client-side web applications. In *ICWE*, pages 186–200, 2013.

[Marchenkov, 1982] S. S. Marchenkov. Unsolvability of positive ∀∃-theory of free semigroup. *Siberian Mathematical Journal*, 36(5):1067–1080, 1982.

[Matiyasevich, 1968] Y. Matiyasevich. The connection between hilberts tenth problem and systems of equations between words and lengths. *Semin. Math., V. A. Steklov Math. Inst., Leningrad*, 36(5):61–67, 1968.

[Matiyasevich, 1997] Yuri Matiyasevich. Some decision problems for traces. In Sergei Adian and Anil Nerode, editors, *Logical Foundations of Computer Science: 4th International Symposium, LFCS'97 Yaroslavl, Russia, July 6–12, 1997 Proceedings*, pages 248–257, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Matiyasevich, 2006] Y. Matiyasevich. Word equations, fibonacci numbers, and hilberts tenth problem, 2006. Available at http://logic.pdmi.ras.ru/~yumat/talks/turku2006.

[McMillan, 2010] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 104–118, Berlin, Heidelberg, 2010. Springer-Verlag.

[Microsoft, 2013] Microsoft. A difficult-to-exploit double free in outlook, 2013. https://blogs.technet.microsoft.com/srd/2013/09/10/ms13-068-a-difficult-to-exploit-double-free-in-outlook/.

[Morgado *et al.*, 2006] Antonio Morgado, Paulo Matos, Vasco Manquinho, and Joao Marques-Silva. Counting models in integer domains. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*, pages 410–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Mozilla, 2016] Mozilla. Javascript reference, 2016. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

[Navarro and Rybalchenko, 2011] P. Navarro and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, 2011.

[Newsome *et al.*, 2006] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *CCS*, pages 311–321, 2006.

[Newsome, 2005] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security Symposium (NDSS)*, 2005.

[Nguyen and Chin, 2008] Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *CAV '08*, pages 355–369. Springer-Verlag, 2008.

[O'Hearn and Pym, 1999] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *BULLETIN OF SYMBOLIC LOGIC*, 5(2):215–244, 1999.

[O'Hearn *et al.*, 2001] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[OWASP, 2013] OWASP. Top ten project, May 2013. http://www.owasp.org/.

[Pek *et al.*, 2014] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *PLDI*, 2014.

[Pieterse and Black, 2004] Vreda Pieterse and Paul E. Black. Dictionary of algorithms and data structures, 2004. https://xlinux.nist.gov/dads/.

[Piskac *et al.*, 2013] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV*, 2013.

[Plandowski, 2006] 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.

[Powell, 2006] Gavin Powell. *Beginning Database Design*, pages 646–662. Wrox Publishing, 2006.

[Qiu *et al.*, 2013] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242. ACM, 2013.

[Quine, 1946] W. V. Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.

[Rakamaric *et al.*, 2007a] Z. Rakamaric, J. D. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI, LNCS 4349*, pages 106–121, 2007.

[Rakamaric *et al.*, 2007b] Z. Rakamaric, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an smt framework. In *ATVA, LNCS 4762*, pages 237–252. Springer, 2007.

[Ranise and Zarba, 2006] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM*, pages 206–215. IEEE-CS, 2006.

[Redelinghuys *et al.*, 2012] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. Symbolic execution of programs with strings. In *SAICSIT*, pages 139–148. ACM, 2012.

[Reynolds, 2000] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[Reynolds, 2002a] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *LICS*. IEEE, 2002.

[Reynolds, 2002b] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[Reynolds, 2003] J. Reynolds. A short course on separation logic. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/aac.html, 2003.

[Rosu and Stefanescu, 2012] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *OOPSLA '12*, pages 555–574. ACM, 2012.

[Roth, 1996] Dan Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302, April 1996.

[Sabelfeld and Myers, 2006] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[Sastry *et al.*, 2006] Naveen Sastry, Tadayoshi Kohno, and David Wagner. Designing voting machines for verification. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[Saxena *et al.*, 2010] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *SP*, pages 513–528, 2010.

[Schulz, 1992] Klaus U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *Proceedings of the First International Workshop on Word Equations and Related Topics*, IWWERT '90, pages 85–150, London, UK, UK, 1992. Springer-Verlag.

[Schwartz *et al.*, 2010] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[Sen *et al.*, 2005] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.

[Sen *et al.*, 2013] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE*, pages 488–498, 2013.

[Shannon *et al.*, 2009] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid. Efficient symbolic execution of strings for validating web applications. In *DEFECTS*, pages 22–26, 2009.

[Smith, 2009] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sonnex *et al.*, 2012] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, 2012.

[Srivastava and Schumann, 2013] Ashok N. Srivastava and Johann Schumann. Software health management: a necessity for safety critical systems. *Innovations in Systems and Software Engineering*, 9(4):219–233, 2013.

[Srivastava, 2010] Saurabh Srivastava. *Satisfiability-Based Program Reasoning and Program Synthesis*. PhD thesis, Department of Computer Science, University of Maryland, 2010.

[Sturton *et al.*, 2009] Cynthia Sturton, Susmit Jha, Sanjit A. Seshia, and David Wagner. On voting machine design for verification and testability. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 463–476, New York, NY, USA, 2009. ACM.

[Tateishi *et al.*, 2011] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA*, pages 166–176, 2011.

[Tateishi *et al.*, 2013] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, pages 33:1–33:33, 2013.

[Trinh *et al.*, 2013] Minh-Thai Trinh, QuangLoc Le, Cristina David, and Wei-Ngan Chin. Bi-abduction with pure properties for specification inference. In *Programming Languages and Systems*, APLAS'13, pages 107–123, 2013.

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

[Trinh *et al.*, 2016] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 218–240, Cham, 2016. Springer International Publishing.

[Tripp *et al.*, 2009] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[Turjan *et al.*, 2002] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in kahn process networks. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, ASAP '02, pages 17–, Washington, DC, USA, 2002. IEEE Computer Society.

[Veanes *et al.*, 2010] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, pages 498–507, 2010.

[Wang *et al.*, 2013] Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen. MetaSymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis. In *Proceedings of the 22nd USENIX Conference on Security*, pages 65–80, 2013.

[Wassermann and Su, 2007] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.

[Wassermann and Su, 2008] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[Wies *et al.*, 2007] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on Heap Abstraction and Verification (collocated with ETAPS)*, 2007.

[Wikipedia, 2016] Wikipedia. C dynamic memory allocation, 2016. https://en.wikipedia.org/wiki/C_dynamic_memory_allocation.

[Xiaobo *et al.*, 2014] Chen Xiaobo, Mike Scott, and Dan Caselden. New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks. Fireeye blog, 2014.

[Xiaofei, 2016] Xie Xiaofei. Static loop analysis and its applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1130–1132, New York, NY, USA, 2016. ACM.

[Xie *et al.*, 2015] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 188–198, New York, NY, USA, 2015. ACM.

[Yu *et al.*, 2010] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. STRANGER: An automata-based string analysis tool for php. In *TACAS*, pages 154–157, 2010.

[Zee *et al.*, 2008] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361. ACM, 2008.

[Zee *et al.*, 2009] K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351. ACM, 2009.

[Zheng *et al.*, 2013] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: a z3-based string solver for web application analysis. In *ESEC/FSE*, pages 114–124, 2013.

[Zheng *et al.*, 2015] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV*. Springer, 2015.