Progressive Reasoning over Recursively-Defined Strings

Minh-Thai Trinh(✉), Duc-Hiep Chu, and Joxan Jaffar

National University of Singapore, Singapore, Singapore
{trinhmt,hiepcd,joxan}@comp.nus.edu.sg

Abstract. We consider the problem of reasoning over an expressive constraint language for unbounded strings. The difficulty comes from "recursively defined" functions such as **replace**, making state-of-the-art algorithms *non-terminating*. Our first contribution is a progressive search algorithm to not only *mitigate* the problem of non-terminating reasoning but also *guide* the search towards a "minimal solution" when the input formula is in fact satisfiable. We have implemented our method using the state-of-the-art Z3 framework. Importantly, we have enabled conflict clause learning for string theory so that our solver can be used effectively in the setting of program verification. Finally, our experimental evaluation shows leadership in a large benchmark suite, and a first deployment for another benchmark suite which requires reasoning about string formulas of a class that has not been solved before.

Keywords: String solving · Progressive search · Termination · Web security

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 [16], 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 [18,21].

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

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

© Springer International Publishing Switzerland 2016
S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 218–240, 2016.
DOI: 10.1007/978-3-319-41528-4_12

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 $X_2 \cdot X_3$. So we replace this substring by Z and then make a recursive call for the remaining part 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 **replace** that are applied to any number of occurrences of a string (even limited to single-character strings) would make the satisfiability problem undecidable [6,7]. 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 [21] 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 [21] 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 [3,12], 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 paper 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 (Sect. 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 (Sect. 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 [18] 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 [19], 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.

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 Fig. 1.

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

Fig. 1. A JavaScript example using **replace** operation

The program takes as its input a JSON [10] 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 Fig. 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:

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

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

$$\begin{gathered}\texttt{res=}\textbf{replace}(\texttt{input,"ip","ip address"}) \wedge \\ \textbf{contains}(\texttt{res,"<script"})\end{gathered}$$

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

$$\begin{gathered}\texttt{input=X}_1\cdot\texttt{"ip"}\cdot\texttt{input}_1 \wedge \texttt{X}_1\cdot\texttt{"i"} \notin /.^\star\ \texttt{"ip"}\ .^\star/ \wedge \\ \texttt{res=X}_1\cdot\texttt{"ip address"}\cdot\texttt{res}_1 \wedge \\ \texttt{res}_1\texttt{=}\textbf{replace}(\texttt{input}_1\texttt{,"ip","ip address"}) \wedge \\ \textbf{contains}(\texttt{res,"<script"})\end{gathered}$$

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

$$\begin{gathered}\texttt{input=X}_1\cdot\texttt{"ip"}\cdot\texttt{input}_1 \wedge \texttt{X}_1\cdot\texttt{"i"} \notin /.^\star\ \texttt{"ip"}\ .^\star/ \wedge \\ \texttt{res=X}_1\cdot\texttt{"ip address"}\cdot\texttt{res}_1 \wedge \texttt{input}_1\texttt{=X}_2\cdot\texttt{"ip"}\cdot\texttt{input}_2 \wedge \\ \texttt{X}_2\cdot\texttt{"i"} \notin /.^\star\ \texttt{"ip"}\ .^\star/ \wedge \texttt{res}_1\texttt{=X}_2\cdot\texttt{"ip address"}\cdot\texttt{res}_2 \wedge \ldots \wedge \\ \texttt{res}_n\texttt{=}\textbf{replace}(\texttt{input}_n\texttt{,"ip","ip address"}) \wedge \\ \textbf{contains}(\texttt{res,"<script"})\end{gathered}$$

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. **replace**, **split**, **match**), we also have non-termination when handling membership predicates with unbounded Kleene-star regular expressions.

Example 1. Unbounded regular expressions:

$$\begin{gathered}\texttt{X=Y}\cdot\texttt{Z}\cdot\texttt{T} \wedge \texttt{Y} \in \texttt{/a}^\star\texttt{/} \wedge \texttt{Z} \in \texttt{/b}^\star\texttt{/} \wedge \texttt{T} \in \texttt{/c}^\star\texttt{/} \wedge \\ \textbf{length}(\texttt{Y})\texttt{=}\textbf{length}\ (\texttt{Z}) \wedge \textbf{length}(\texttt{Z})=\textbf{length}\ (\texttt{T}) \wedge \texttt{X=X}_1\cdot\texttt{"d"}\cdot\texttt{X}_2\end{gathered}$$

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 [21,23] represent the membership predicate $\texttt{X}{\in}\texttt{/a}^{\star}\texttt{/}$ as an equation between string variable X and **star**(a,N) function which can be defined recursively as below:

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

To facilitate the solving process, [21,23] 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 Sect. 4.

Finally, we note that the problem of non-terminating reasoning is not solely due to the recursive definitions we employ in this paper. 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:

$$\texttt{X} \cdot \texttt{"a"} = \texttt{"b"} \cdot \texttt{X}$$

The classic work [15] 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 $\texttt{X}{\cdot}\texttt{"a"=}\texttt{"b"}{\cdot}\texttt{X} \wedge \texttt{X=}\texttt{"b"}{\cdot}\texttt{Y}$. This can then be simplified into $\texttt{Y}{\cdot}\texttt{"a"=}\texttt{"b"}{\cdot}\texttt{Y} \wedge \texttt{X=}\texttt{"b"}{\cdot}\texttt{Y}$. Notice that the last formula is, in some sense, equally difficult to solve as the original one. The huge contribution of [15] 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.

3 The Core Language

We introduce the core constraint language in Fig. 2. In our implementation, the string theory solver is a component of Z3 solver [9]. Though Z3 supports more primitive types, we only mention string type and integer type in Fig. 2.

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

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

Fig. 2. The syntax of our core constraint language

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 **replace**, 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 **replace** and **star**.

Following [21], 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 [21], 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 [21,23], 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 Fig. 2.

4 Algorithm

In Sect. 4.1, we first present the background and limitation of existing methods. In Sect. 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 Sect. 5.

4.1 Preliminaries

This paper builds on top of the string solver S3 [21]. Essentially, the S3 solver is a string theory plug-in built into the Z3 SMT solver [9], 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 [21,23].

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

Definition 1 (Derivation Rule). *Each rule is of the general form*

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

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

[1] As per Fig. 2.

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 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.

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 paper. 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 Figs. 3 and 4;
- multi-reduct rules: in Fig. 5.

We first describe the one-reduct rules in Fig. 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 `Var`(F), `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.

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

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

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

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

Fig. 3. Length constraint propagation rules

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.

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

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

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

Fig. 4. Simplification rules for string constraints

Next, consider Fig. 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.

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 Fig. 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 Figs. 3 and 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.

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

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

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

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

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

Fig. 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 `replace` function presented in Sect. 2). We describe such a rule using an unfolding rule *schema* (UNF) for a recursive function E as follows:

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

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:

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

The specific instances of (SPL) and (UNF) rules used in this paper are listed in Fig. 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 Examples 3 and 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 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 **star**("a", N) function and normalize the formula to DNF. An application of the unfold rule (UNF-⋆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:

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

This contradiction appears in the tree depicted in Fig. 6, but is hidden in the part of the tree that was abbreviated away for brevity.

In the second reduct, by substituting X with $Y_1{\cdot}$ "b" ${\cdot}Z$, 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.

$$\text{(UNF-}\star\text{1)}\ \frac{X = \mathbf{star}(``a", N) \wedge X = Y_1 \cdot ``b" \cdot Z}{X{=}``a"{\cdot}\mathbf{star}(``a", M) \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}``b"{\cdot}Z}$$

$$\vdots$$

$$\text{(SUB)}\ \frac{X{=}``a"{\cdot}\mathbf{star}(``a", M) \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}``b"{\cdot}Z}{Y_1{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", M){\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}$$

$$\text{(SPL-2)}\ \frac{Y_1{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", M){\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}{Y_1{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", M){\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}$$

$$\text{(SUB)}\ \frac{Y_1{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", M){\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}{``a"{\cdot}Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a",M){\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}$$

$$\text{(SIM)}\ \frac{``a"{\cdot}Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a",M){\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z}{Y_2{\cdot}``b"{\cdot}Z{=}\mathbf{star}(``a", M) \wedge Y_1{=}``a"{\cdot}Y_2 \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}``b"{\cdot}Z}$$

$$\text{(UNF-}\star\text{1)}\ \frac{Y_2{\cdot}``b"{\cdot}Z{=}\mathbf{star}(``a", M) \wedge Y_1{=}``a"{\cdot}Y_2 \wedge N{=}M{+}1 \wedge X{=}Y_1{\cdot}``b"{\cdot}Z}{\begin{array}{c} Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}$$

$$\vdots$$

$$\text{(SPL-2)}\ \frac{\begin{array}{c} Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}{\begin{array}{c} Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}$$

$$\text{(SUB)}\ \frac{\begin{array}{c} Y_2{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}{\begin{array}{c} ``a"{\cdot}Y_3{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}$$

$$\text{(SIM)}\ \frac{\begin{array}{c} ``a"{\cdot}Y_3{\cdot}``b"{\cdot}Z{=}``a"{\cdot}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}{\begin{array}{c} Y_3{\cdot}``b"{\cdot}Z{=}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}$$

$$\text{(UNF-}\star\text{1)}\ \frac{\begin{array}{c} Y_3{\cdot}``b"{\cdot}Z{=}\mathbf{star}(``a", P){\wedge}Y_2{=}``a"{\cdot}Y_3{\wedge} \\ M{=}P{+}1{\wedge}Y_1{=}``a"{\cdot}Y_2{\wedge}N{=}M{+}1{\wedge}X{=}Y_1{\cdot}``b"{\cdot}Z \end{array}}{\vdots}$$

Fig. 6. Derivation tree for Example 3

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 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* `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 $\mathtt{Len}_\sigma(\alpha)$ 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 (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 three important properties[2] of $\prec_\sigma$:

- [Prop-1]: If $F \equiv (G \vee H)$ where $\texttt{Var}(F) \subseteq \texttt{Var}(H)$ and $\exists\sigma : F \prec_\sigma G$ then $F =_\sigma H$
- [Prop-2]: If $(G \vee H) \Rightarrow F$ and $\exists\sigma : F =_\sigma G$ then $F =_\sigma (G \vee H)$
- [Prop-3]: If $\exists\sigma : F =_\sigma G$ and τ is a prefix of σ then $F =_\tau G$

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.

```
Input: $I$ : $Fml$ , $\tau$ : a sequence on $\texttt{Var}(I)$
Output: SAT/UNSAT
⟨1⟩  if SOLVE($I$, $\tau$, $\emptyset$)  return SAT  else return UNSAT

function SOLVE($H$ : $Fml$, $\sigma_I$: a sequence, $\gamma$: a list of pairs of a formula and a sequence)
⟨2⟩   if ($H \equiv$ 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 $\Upsilon$ 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 $\sigma_H$ be a sequence on $\texttt{Var}(H)$ s.t. $\sigma_I$ is a prefix of $\sigma_H$ /* CONDITION 1 */
⟨10⟩            $\gamma \leftarrow \gamma \cup \langle H, \sigma_H \rangle$
⟨11⟩       endif
⟨12⟩       if SOLVE($G$, $\sigma_I$, $\gamma$) return true
⟨13⟩       if ($G$ contains a recursive term or a non-grounded concatenation)
⟨14⟩            $\gamma \leftarrow \gamma \setminus \{H, \sigma_H\}$
⟨15⟩  endfor
⟨16⟩  return false
end function
```

Algorithm S3P. Progressive Search

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 Sect. 4.1 with a new step which enables us to prune the proof tree.

Definition 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:*

[2] All the proofs are in our technical report [22].

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

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, we still preserve a minimal solution of formula F w.r.t. a sequence of all of the variables of F.

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 subformulas; 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 5 is met).

Theorem 1 (Soundness). *Given an input formula I, if Algorithm S3P*

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

□

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 Sect. 5.1, we present an implementation which, though not a decision procedure, is sound and practical. We follow this up in Sect. 6 with an experimental evaluation.

Theorem 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.* □

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.

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:

$$\tau \text{ is } (N, X, Y_1, Z) \text{ and } \gamma \text{ is } \{(X = \textbf{star}(``a", N) \wedge X = Y_1 \cdot ``b" \cdot Z,\ \tau)\}.$$

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 6 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 $\texttt{false}$ trivially.

Definition 6 (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 $\texttt{Len}_\sigma(\beta) < \texttt{Len}_\sigma(\alpha)$.* $\square$

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

$$G \equiv Y_2 \cdot ``b" \cdot Z = \textbf{star}(``a", M) \wedge Y_1 = ``a" \cdot Y_2 \wedge N = M + 1 \wedge X = Y_1 \cdot ``b" \cdot Z$$

Obviously, there exists $F \equiv X = \textbf{star}(``a", N) \wedge X = Y_1 \cdot ``b" \cdot Z$ and a substitution $\theta = [M/N, N/N+1, X/``a" \cdot X, Y_1/``a" \cdot Y_1, Y_2/Y_1, Z/Z]$, 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 $\textbf{length}(N) = k$ in a solution α (if any) of G, we have $\textbf{length}(M) = k - 1$. Then, we have $\textbf{length}(N) = k - 1$ in the corresponding solution α' of $G\theta$. Because $\texttt{Len}_\tau$ function returns a 4-tuple whose first element is $\textbf{length}(N)$, θ is progressive. As a result, we can stop the second unfolding.

Lemma 1. *The implementation of the pruning step is sound* $\square$

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 1 is that we have $F \prec_\sigma G$ (see the proof of Lemma 1 in [22]). Now, in addition to returning $\texttt{false}$

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 Lemmas 2 and 3.

Definition 7 (String Solved Form). *A string equation is in solved form if it is in the form of* $X = f(Y_1, \ldots, Y_n, a_1, \ldots, a_m)$, *where* $X \in V_{str}$, $Y_1, \ldots, Y_n \in V_{str}$, $a_1, \ldots, a_n \in C_{str}$, $X \notin \{Y_1, \ldots, Y_n\}$, *and f is a non-recursive function.* □

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

Definition 8 (Integer Solved Form). *An equation is in solved form if it is in the form of* $M = g(N_1, .., N_n, p_1, \ldots, p_m)$, *where* $M \in V_{int}$, $V_1, .., V_n \in V_{int} \cup V_{str}$, $p_1, \ldots, p_m \in C_{int} \cup C_{str}$, $M \notin \{N_1, \ldots, N_n\}$, *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 $Y_2 \cdot \text{“}b\text{”} \cdot Z = \textbf{star}(\text{“}a\text{”}, M) \wedge Y_1 = \text{“}a\text{”} \cdot Y_2 \wedge N = M + 1 \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$. 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 $\widehat{G} \equiv Y_2 \cdot \text{“}b\text{”} \cdot Z = \textbf{star}(\text{“}a\text{”}, M)$ 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 $X = \textbf{star}(\text{“}a\text{”}, N) \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$.

Lemma 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.* □

Lemma 3. *$\widehat{G}$ is satisfiable iff G is satisfiable.* □

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 $Y_2 \cdot \text{“}b\text{”} \cdot Z = \textbf{star}(\text{“}a\text{”}, M) \wedge Y_1 = \text{“}a\text{”} \cdot Y_2 \wedge N = M + 1 \wedge X = Y_1 \cdot \text{“}b\text{”} \cdot Z$, we have 4 equivalence classes as follows:

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

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 $\textbf{length}(D)$, $\textbf{concat}(D, E)$). 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 $\widehat{G} \equiv Y_2 \cdot \text{“}b\text{”} \cdot Z = \textbf{star}(\text{“}a\text{”}, M)$.

6 Evaluation

We implemented our algorithm into S3 [21] which itself was built on top of the Z3 framework [9]. 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 3.2 GHz 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 [18]. 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 **replace** have been transformed to primitive operators so that the underlying solver of Kudzu [18] can handle. Because strong support for the **replace** 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 **replace** 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 [18], a symbolic execution framework for JavaScript,

Table 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

Table 2. Usefulness of unsatisfiable cores for Kudzu framework

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

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 [14].

This case study consists of two parts. The first part is to evaluate our non-progression detection technique. Table 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.

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" [23].

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.

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 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 **replace** string function. As such, we collect constraints generated by testing web applications using the concolic tester in Jalangi framework [19], 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 3. Constraints generated by Jalangi

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

Table 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 **replace** operation. So the percentage of **replace** operation is about 15 %.

More importantly, **replace** 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 **replace** 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:

$$\texttt{input}_1 = \textbf{replace}(\texttt{input}, \text{“}<\text{”}, \text{“}\texttt{\&lt;}\text{”}) \land\ \texttt{input}_2 = \textbf{replace}(\texttt{input}_1, \text{“}>\text{”}, \text{“}\texttt{\&gt;}\text{”}) \ \land \\ \texttt{output} = \texttt{input}_2 \ \cdot\ \text{“}</\texttt{br}>\text{”} \ \land \textbf{contains}(\texttt{output}, \text{“}<\texttt{script}\text{”})$$

Given that Z3-str2, CVC4, and Norn *cannot* deal with **replace** 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 Sect. 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.

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 [13], CFGAnalyzer [4,11]) 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 [18], 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 [6,18] is to proceed by first enumerating concrete length values, before encoding strings into bit-vectors. In a similar manner, [17] 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 [21].

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) [8] 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 [20] 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 [5] provides an automata-based technique for solving string constraints, and further, a method for counting the number of solutions to such constraints. A recent string solver Norn [1,2] is also based on automata techniques.

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 paper. 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 work on string solving are based on unbounded methods with string as a primitive data type. Examples are Z3-str [24], CVC4 [14], S3 [21], Z3-str2 [23]. However, none of them addresses the non-termination issues in string solving as in this paper. Though in [23], 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.

8 Conclusion

This paper 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.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Heidelberg (2014)
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Heidelberg (2015)
3. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: ICSE, pp. 1083–1094. ACM (2014)
4. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
5. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Heidelberg (2015)
6. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
7. Buchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. In: Mathematical Logic Quarterly, pp. 337–342 (1988)
8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: SAS, pp. 1–18 (2003)
9. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. ECMA-404. Javascript object notation. http://www.json.org/
11. He, J., Flener, P., Pearson, J., Zhang, W.: Solving string constraints: the case for constraint programming. In: CP, pp. 381–397 (2013)
12. Jaffar, J., Murali, V., Navas, J.A.: Boosting concolic testing via interpolation. In: FSE, pp. 48–58. ACM (2013)
13. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: ISSTA, pp. 105–116. ACM (2009)
14. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Heidelberg (2014)
15. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. USSR-Sbornik **32**(2), 129 (1977)
16. OWASP. Top ten project, May 2013. http://www.owasp.org/
17. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: SAICSIT, pp. 139–148. ACM (2012)
18. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: SP, pp. 513–528 (2010)
19. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for javascript. In: FSE, pp. 488–498 (2013)
20. Shannon, D., Ghosh, I., Rajan, S., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: DEFECTS, pp. 22–26 (2009)

21. Trinh, M.-T., Chu, D.-H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in webapplications. In: ACM-CCS, pp. 1232–1243. ACM (2014)
22. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. Technical report (2016). http://www.comp.nus.edu.sg/~trinhmt/progressive/
23. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 235–254. Springer, Heidelberg (2015)
24. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: ESEC/FSE, pp. 114–124 (2013)