A Decision Procedure for String Constraints with String-Integer Conversion and Flat Regular Constraints

Hao Wu[1,2], Yu-Fang Chen[3], Zhilin Wu[1,2], Bican Xia[4] and Naijun Zhan[1,2*]

[1*]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.
[2]University of Chinese Academy of Sciences, Beijing, China.
[3]Institute of Information Science, Academia Sinica, Taiwan, Republic of China.
[4]School of Mathematical Sciences, Peking University, Beijing, China.

*Corresponding author(s). E-mail(s): znj@ios.ac.cn;
Contributing authors: wuhao@ios.ac.cn; yfc@iis.sinica.edu.tw; wuzl@ios.ac; xbc@math.pku.edu.cn;

Abstract

String constraint solving is the core of various testing and verification approaches for scripting languages. Among algorithms for solving string constraints, flattening is a well-known approach that is particularly useful in handling satisfiable instances. As string-integer conversion is an important function appearing in almost all scripting languages, Abdulla et al. extended the flattening approach to this function recently. However, their approach supports only a special flattening pattern and leaves the support of the general flat regular constraints as an open problem. In this paper, we fill the gap by proposing a complete flattening approach for the string-integer conversion. The approach is built upon a new quantifier elimination procedure for the linear-exponential arithmetic (namely, the extension of Presburger arithmetic with exponential functions, denoted by ExpPA) improved from the one proposed by Cherlin and Point in 1986. We analyze the complexity of our quantifier elimination procedure and show that the decision problem for existential ExpPA formulas is in 3-EXPTIME. Up to our knowledge, this is the first elementary complexity upper bound for this problem. While the quantifier elimination

procedure is too expensive to be implemented efficiently, we propose various optimizations and provide a prototypical implementation. We evaluate the performance of our implementation on the benchmarks that are generated from the string hash functions as well as randomly. The experimental results show that our implementation outperforms the state-of-the-art solvers.

Keywords: String-integer conversion, flat regular constraints, exponential function, Presburger arithmetic, quantifier elimination

1 Introduction

The emerging of scripting languages boosts the needs of efficient approaches and tools to ensure program quality. Comparing with traditional programming languages, string data type plays a more critical role in its analysis. String constraint solvers are the engine of modern scripting program analysis techniques. Due to the high demand, in recent years, there is a boosting amount of publications on this subject.

However, research progress of string constraint solving has been hampered by many major challenges in both theory and tool implementation aspects (including long-standing open problems). Logical theories over strings have to allow string concatenation, which is arguably the most fundamental operation of strings. The most celebrated result concerning theories of strings is Makanin's result on deciding the satisfiability problem for *word equations* [1]. A simple example of a word equation is $xaby = ybax$, where x, y are variables, and $a, b \in \Sigma$ are constant letters. A word equation is satisfiable if it has a solution, i.e., an assignment that maps variables to strings over the alphabet Σ which equates the left-hand side with the right-hand side of the equation. The correctness proof of Makanin's algorithm is arguably one of the most complex termination proofs in computer science. Makanin's result can be extended to include *regular constraints* (a.k.a. regular expression matching, e.g., $x \in (ba)^*$), and arbitrary Boolean connectives. This extension is called word equations with regular constraints. However, the satisfiability problem of word equations together with length constraints (e.g., $|x| = |y|+1 \wedge wx = yx$) is still open. The complexity of the satisfiability of word equations with regular constraints was proven to be PSPACE-complete by Plandowski [2], after decades of improvement of the original algorithm by Makanin.

Satisfiability of word equations is a special instance of Hilbert's 10th problem. In the past, the original motivation of studying word equations was to find an undecidability proof of Hilbert's 10th problem. However, the motivation is no longer valid since Makanin finds a decision procedure. Recently, driven by the need for program analysis, people started to revisit the problem and its extensions to describe the complete string library APIs in conventional programming languages. Many highly efficient solvers for string constraints, to name a few, CVC4 [3], Z3 [4], Z3-Str3 [5], S3 [6], Norn [7], Ostrich [8], Sloth [9], ABC [10], Stranger [11], Trau [12] and so on, are developed in the last decade. The satisfiability of *string-integer conversion constraints*, e.g., $wx = yx \wedge |x| > \mathsf{parseInt}(y)$, has been proven undecidable

in [13]. However, this kind of constraints is pervasive in scripting language programs. For example, it is common that programs read string inputs from text files and converts a part of the string input to integers. Even more crucially, in many programming languages, the string-integer conversion is a part of the definition of their core semantics [14]. JavaScript, which powers most interactive content on the Web and increasingly server-side code with Node.js, is one of such languages.

Due to the inherent difficulties and practical importance of solving string constraints, one idea is to have separate specialized procedures for solving satisfiable sub-problems. Currently, there are two main specialized approaches for proving satisfiability. The first is to consider only strings of bounded length. This approach is taken in the first-generation solvers such as Hampi [15] and Kuluza [16]. Although they are useful in handling many practical cases, they fail to find an answer when all string solutions exceed the selected bound. For example, a constraint of the form $xy \neq z \wedge |x| > 2000$ would be quite challenging using those solvers.

One more recent approach is flattening [12, 17, 18]. The idea is to restrict the solution space of string variables to *(parametric) flat languages* (see Section 3). The major benefit of considering this class is two-fold. First, under the restriction, the potential solution space is still infinite, which gives us a higher potential of finding solutions. For instance, we can find a solution for $xy \neq z \wedge |x| > 2000$ under a very simple restriction: all variables are in a^*, where a is an element of the alphabet. Second, more importantly, because we can convert the membership problem of a flat language to the satisfiability problem of a Presburger arithmetic formula, the class of word equations + flat languages + length constraints is decidable.

The paper of Abdulla et al. [17] considered adding string-integer conversion constraints to the above class, and proposed an algorithm for a restricted form of flat languages and left the support of general flat languages as an open problem. For string-integer conversion constraints, their approach projects the solution to a finite solution space (in a way similar to the PASS approach [19]).

In this paper, we give a complete solution to this problem. We propose a decision procedure for the class of word equations + flat languages + length constraints + string-integer conversion. The basic idea of our approach can be sketched as follows: we first reduce the satisfiability problem to the corresponding satisfiability problem of ExpPA, more precisely, the existential fragment of ExpPA; then, according to the decidability of ExpPA, we obtain the decidability of the original satisfiability problem.

The decidability of ExpPA was first shown by Semenöv in [20]. Nevertheless, Semenöv did not provide an explicit decision procedure. To remedy this, in [21], Cherlin and Point presented the first quantifier elimination procedure for the satisfiability of ExpPA; later Point revisited the procedure in [22]. Partially attributed to its non-elementary complexity, this quantifier elimination procedure has mostly eluded the attentions of computer science community. To the best of our knowledge, no implementation based on Cherlin and Point's procedure was available up to now.

Aiming at utilizing Cherlin and Point's procedure in practice, we reformulate the underlying theory ExpPA and improve the algorithm by incorporating Cooper's quantifier elimination procedure for PA, which is another main contribution of this paper

(Section 4). We analyze the complexity of our improved algorithm and show that existentially quantified ExpPA formulas are decidable in 3-EXPTIME (Section 5). To the best of our knowledge, this is the first elementary complexity upper bound for decision problems in ExpPA. Furthermore, we propose various optimizations (Section 6) and achieve the first prototypical implementation (Section 7).

In fact, other than the theoretical difficulties, in practice, the string-integer conversion is quite challenging for state-of-the-art solvers. Here we illustrate a toy example that mimic the "mining" step of block-chain construction. Essentially, given a string hash function $\mathsf{hash} : \Sigma^* \to \mathbb{N}$, the goal of the mining step is to find a string v such that when inserting v into the text to be protected, say w_1 and w_2, the hash value $\mathsf{hash}(w_1 \cdot v \cdot w_2)$ satisfies a certain pattern, e.g., the last k digits are zeros. Here, the string v is also called a *nonce* for it is used only once. If w_1 or w_2 are modified, one needs to compute another v which satisfies the desired pattern. Below we consider a simple hash function: $\mathsf{hash}(w) = \sum_{i=1}^{n} a_i p^{n-i} \bmod m$, where $w = a_1 \dots a_n \in \Sigma^*$ with $\Sigma \subseteq \mathbb{N}$ and $p, m \in \mathbb{Z}^+$ are user-defined numbers. It is easy to see that $\mathsf{hash}(w)$ can be seen as a generalization of $\mathsf{parseInt}$ followed by a modulo operation. In particular, if $\Sigma = \{0, 1, \dots, 9\}$ and $p = 10$, then $\mathsf{hash}(w) = \mathsf{parseInt}(w) \bmod m$. Thus, the problem of finding a suitable input w such that the last k digits of $\mathsf{hash}(w)$ are zeros can be modeled as a string constraint with $\mathsf{parseInt}$. Although the example is seemingly simple, it is already challenging for most state-of-the-art solvers, as shown by our experiment results in Section 7. With the optimizations introduced in Section 6, our implementation manages to solve several variants of the string-hash examples as well as some randomly generated arithmetic problem instances better than the state-of-the-art solvers (Section 7).

Structure

After introducing preliminaries in Section 2, we present how to flatten a string constraint with string-integer conversion to an existential ExpPA formula in Section 3. We describe the quantifier elimination procedure for ExpPA in Section 4 and analyze the complexity of deciding existential ExpPA formulas in Section 5. Several optimization techniques are provided in Section 6. Finally, we report the implementation and experiment results in Section 7.

2 Preliminaries

In this section, we fix the notations and introduce some basic concepts, including Presburger arithmetic, finite-state automata, and flat languages.

Integers, strings, and languages

Let $\mathbb{N}$ denote the set of natural numbers, $\mathbb{Z}$ denote the set of integers, and $\mathbb{Z}^+$ denote the set of positive integers. For $n \in \mathbb{Z}^+$, let $[n]$ denote the set $\{1, \dots, n\}$.

An *alphabet* Σ is a finite set. Each element of Σ is called a *letter*. A *string* w over Σ is a (possibly empty) finite sequence $a_1 \dots a_n$ with $a_i \in \Sigma$ for every $i \in [n]$. Let ε denote the empty string, namely, the empty sequence. Let Σ^* denote the set

of all strings over Σ. Let Σ^+ denote the set of nonempty strings over Σ. For convenience, we also use Σ_ϵ to denote $\Sigma \cup \{\epsilon\}$. For a string $w = a_1 \dots a_n \in \Sigma^*$, let $\mathsf{len}(w)$ denote the *length* of w, i.e., n. In particular, $\mathsf{len}(\varepsilon) = 0$. For $w_1 = a_1 \dots a_m, w_2 = b_1 \dots b_n \in \Sigma^*$, let $w_1 \cdot w_2$ denote the *concatenation* of w_1 and w_2, that is, $a_1 \dots a_m b_1 \dots b_n$. A language L over Σ is a subset of Σ^*.

Presburger Arithmetic

Presburger Arithmetic (PA) is the first-order logic of integers with addition. A term of PA, denoted by $\mathtt{t}$, is of the form

$$\mathtt{t} \mathrel{\hat{=}} c \mid \mathbb{x} \mid \mathtt{t} + \mathtt{t} \mid \mathtt{t} - \mathtt{t},$$

where $\mathbb{x}$ and c represent integer variables and integer constants respectively. A formula of PA, denoted by ϕ, is of the form

$$\phi \mathrel{\hat{=}} \mathtt{t} \odot \mathtt{t} \mid c \diamond \mathtt{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists \mathbb{x}.\ \phi \mid \forall \mathbb{x}.\ \phi,$$

where $\odot \in \{=, <, >, \leq, \geq\}, \diamond \in \{|, \nmid\}$.

An occurrence of a variable $\mathbb{x}$ in ϕ is *bounded* if it occurs in a subformula of ϕ with the form $\exists \mathbb{x}.\phi'$ or $\forall \mathbb{x}.\phi'$; otherwise, it is called *free* in ϕ. A variable of ϕ is said to be *free* if it has free occurrences, and *bounded* otherwise. The set of free variables of ϕ is denoted by $\mathrm{Free}(\phi)$. We usually write $\phi(\mathbb{x}_1, \cdots, \mathbb{x}_k)$ to denote an PA formula ϕ such that $\mathrm{Free}(\phi) \subseteq \{\mathbb{x}_1, \cdots, \mathbb{x}_k\}$. Given an PA formula ϕ and an integer interpretation of $\mathrm{Free}(\phi)$, i.e., a function $I : \mathrm{Free}(\phi) \to \mathbb{Z}$, we denote by $I \models \phi$ that I satisfies ϕ (which is defined in the standard manner, with $+$, $-$, $|$ and $\nmid$ interpreted as the integer addition, subtraction, divisibility and indivisibility relation respectively), and call I a *model* of ϕ. We use $[\![\phi]\!]$ to denote the set of models of ϕ. By convention, we write $I \not\models \phi$ if $I \models \neg\phi$.

A *quantifier-free* PA (QFPA) formula is a PA formula containing no quantifiers. We say a PA formula is in *prenex normal form* (PNF) if it has the form $Q_1 \mathbb{x}_1 \dots Q_n \mathbb{x}_n.\varphi$ with $Q_i \in \{\exists, \forall\}$ for $i \in [n]$ and $\varphi \in \mathsf{QFPA}$. A PA formula is called *existential* if it is in PNF and contains no occurrences of universal quantifiers.

In the rest of the paper, we assume, without loss of expressiveness, that QFPA formulas contain no negations, because a formula can be first equivalently transformed into negation normal form, and negations before atomic formulas can be then absorbed by changing the predicates (for example, $\neg(\mathbb{x} \geq \mathbb{y}) \equiv \mathbb{x} < \mathbb{y}$). The only cases where we deal with negations is to transform an universal quantifier $\forall$ into an existential one $\neg\exists\neg$.

It is well-known that PA admits quantifier elimination, for example, Cooper's algorithm [23]. In Section 4, we will use this fact and apply Cooper's algorithm as a subprocedure.

Finite state automata

A *finite state automaton* (FA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, q_{init}, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma_\epsilon \times Q$ is the transition relation, q_{init}

is the initial state, and $F \subseteq Q$ is the set of accepting states. A *run* of $\mathcal{A}$ on a string $w = a_1 \dots a_n$ is a sequence $q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_{m-1}} q_{m-1} \xrightarrow{b_m} q_m$ such that $q_0 = q_{init}$, $(q_{i-1}, b_i, q_i) \in \Delta$ for every $i \in [m]$, and $a_1 \dots a_n = b_1 \dots b_m$ (in general, $m \geq n$ since b_i may be ϵ). We say a run $q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_{m-1}} q_{m-1} \xrightarrow{b_m} q_m$ is *accepting* if $q_m \in F$. A string w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. Let $\mathcal{L}(\mathcal{A})$ denote the set of strings accepted by $\mathcal{A}$. A language $L \subseteq \Sigma^*$ is *regular* if it can be defined by some FA $\mathcal{A}$, namely, $L = \mathcal{L}(\mathcal{A})$.

Flat languages and parametric flat languages

We now define flat languages and parametric flat languages from a high level, language-theoretical view [17].

A *flat language* (FL) over Σ is the set of strings that conform to a regular expression of the form $v_1(w_1)^* v_2(w_2)^* \dots v_n(w_n^*)v_{n+1}$, where $w_i, v_j \in \Sigma^*$ for $i \in [n], j \in [n+1]$. Intuitively, an FL is a sequence of loops connected by finite strings.

A *parametric flat language* (PFL) is a FL defined on an alphabet V of *character variables*. Specifically, a PFL is of the form $v_1(w_1)^* v_2(w_2)^* \dots v_n(w_n^*)v_{n+1}$, where $w_i = \mathfrak{a}_1^i \cdots \mathfrak{a}_{k_i}^i$ and $v_j = \mathfrak{b}_1^j \cdots \mathfrak{b}_{k'_j}^j$ with $\mathfrak{a}_{\_}^i, \mathfrak{b}_{\_}^j \in V$. Moreover, it is required that all $\mathfrak{a}_{\_}^i$ and $\mathfrak{b}_{\_}^j$ represent distinct elements in V. An *interpretation* of V is a mapping $I_p : V \to \Sigma_\epsilon$. Given a PFL $\mathfrak{L}$ and an interpretation I_p, one can translate $\mathfrak{L}$ into a FL over Σ_ϵ, denoted by $I_p(\mathfrak{L})$.

Example 1 The language defined by $\mathfrak{b}_1^1\mathfrak{b}_2^1(\mathfrak{a}_1^1\mathfrak{a}_2^1)^*\mathfrak{b}_1^2(\mathfrak{a}_1^2\mathfrak{a}_2^2)^*$ is a PFL. When given the interpretation $I_p = \{\mathfrak{b}_1^1 \mapsto 1, \mathfrak{b}_2^1 \mapsto 1, \mathfrak{a}_1^1 \mapsto 1, \mathfrak{a}_2^1 \mapsto 0, \mathfrak{b}_1^2 \mapsto \epsilon, \mathfrak{a}_1^2 \mapsto 0, \mathfrak{a}_2^2 \mapsto 0\}$, one can translate the PFL into the FL $11(10)^*(00)^*$.

3 Flattening string constraints with parseInt

In this section, we first define the class of string constraints with string-integer conversion, denoted by $\mathsf{STR}_{\mathsf{parseInt}}$. Then we define the extension of Presburger arithmetic with exponential functions, denoted by ExpPA. Finally, we show how to flatten the string constraints in $\mathsf{STR}_{\mathsf{parseInt}}$ into the arithmetic constraints in the existential fragment of ExpPA.

3.1 String constraints with string-integer conversion ($\mathsf{STR}_{\mathsf{parseInt}}$)

In the sequel, we shall define $\mathsf{STR}_{\mathsf{parseInt}}$, the class of string constraints with the string-integer conversion function parseInt.

The function parseInt takes a decimal string as input and returns the integer represented by the string. For example,

$$\mathsf{parseInt}(\text{`0123'}) = \mathsf{parseInt}(\text{`123'}) = 1 * 10^2 + 2 * 10 + 3 = 123$$

where we use the quotation marks to delimit strings. One should note that, in scripting languages like Javascript, parseInt is more general in the sense that the base can be a number between 2 and 36. Although our approach works for arbitrary positive bases, we choose to focus on the base 10 in this paper for simplicity.

Formally, the semantics of the parseInt function is defined as follows. In order to simplify the presentation, we assume all string variables ranging over numerical symbols $\Sigma_{num} = \{0, 1, \ldots, 9\}$. Note that one can easily extend our approach to allow arbitrary finite alphabet. Then $\mathsf{parseInt} : \Sigma_{num}^{+} \mapsto \mathbb{N}$ is recursively defined by, for every $w \in \Sigma_{num}^{+}$,

- if $w =$‘i’ for $i \in \Sigma_{num}$, then $\mathsf{parseInt}(\text{‘}i\text{’}) = i$;
- if $w = w'\cdot$‘i’ for $i \in \Sigma_{num}$ with $\mathsf{len}(w') \geq 1$, $\mathsf{parseInt}(w) = 10 * \mathsf{parseInt}(w') + \mathsf{parseInt}(\text{‘}i\text{’})$.

Note that parseInt is undefined with ε as the input.

In $\mathsf{STR}_{\mathsf{parseInt}}$, there are two types of variables, i.e., the string variables $x, y, \ldots \in \mathcal{X}$ and the integer variables $\mathbb{x}, \mathbb{y}, \ldots \in \mathbb{X}$. The syntax of $\mathsf{STR}_{\mathsf{parseInt}}$ is defined as follows: a string term, denoted by t, is of the form

$$t \hat{=} a \mid x \mid t \cdot t,$$

an integer term, denoted by $\mathbb{t}$, is of the form

$$\mathbb{t} \hat{=} c \mid \mathbb{x} \mid \mathsf{len}(t) \mid \mathsf{parseInt}(t) \mid \mathbb{t} + \mathbb{t} \mid \mathbb{t} - \mathbb{t},$$

and a $\mathsf{STR}_{\mathsf{parseInt}}$ formula, denoted by φ is of the form

$$\varphi \hat{=} t = t \mid t \in \mathcal{A} \mid \mathbb{t} \odot \mathbb{t} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$$

where $a \in (\Sigma_{num})_{\varepsilon}$, $c \in \mathbb{Z}$, $\mathcal{A}$ is an FA, $\odot \in \{=, <, >, \leq, \geq\}$ and $\mathsf{len}(t)$ denotes the length of a string t. Let us call $t = t$ as string equality constraints, $t \in \mathcal{A}$ as regular constraints, $\mathbb{t} \odot \mathbb{t}$ as arithmetic constraints. Let $\mathsf{SVar}(\varphi)$ and $\mathsf{IVar}(\varphi)$ denote the set of string variables and integer variables occurring in φ respectively.

An interpretation I over $\mathcal{X}$ and $\mathbb{X}$ is a tuple $I \hat{=} (I_S, I_N)$ with $I_S : \mathcal{X} \to \Sigma_{num}^{*}$ and $I_N : \mathbb{X} \to \mathbb{Z}$. We lift the interpretation $I = (I_S, I_N)$ to string terms and linear terms in the standard manner. For a $\mathsf{STR}_{\mathsf{parseInt}}$ formula φ, let $[\![\varphi]\!]$ denote the set of all interpretations that satisfy φ. $[\![\varphi]\!]$ is defined inductively as follows:

$$\begin{aligned}
[\![t_1 = t_2]\!] &\hat{=} \{I = (I_S, I_N) \mid I_S(t_1) = I_S(t_2)\} \\
[\![t \in \mathcal{A}]\!] &\hat{=} \{I = (I_S, I_N) \mid I_S(t) \in \mathcal{L}(\mathcal{A})\} \\
[\![\mathbb{t}_1 \odot \mathbb{t}_2]\!] &\hat{=} \{I = (I_S, I_N) \mid I_N(\mathbb{t}_1) \odot I_N(\mathbb{t}_2)\} \\
[\![\varphi_1 \wedge \varphi_2]\!] &\hat{=} [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\varphi_1 \vee \varphi_2]\!] &\hat{=} [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \\
[\![\neg\varphi]\!] &\hat{=} ([\![\varphi]\!])^{C}
\end{aligned}$$

The *satisfiability problem* of $\mathsf{STR}_{\mathsf{parseInt}}$ is to decide, for a given constraint $\varphi \in \mathsf{STR}_{\mathsf{parseInt}}$, whether $[\![\varphi]\!]$ is nonempty.

Example 2 Given a FA $\mathcal{A}$, the constraint

$$x \in \mathcal{A} \ \wedge \ \mathsf{parseInt}(x) = 109\mathbb{x} \ \wedge \ \mathsf{len}(x) < 100$$

is a $\mathsf{STR}_{\mathsf{parseInt}}$ formula.

3.2 Presburger Arithmetic with exponential functions (ExpPA)

ExpPA extends Presburger arithmetic with two partial functions, the *exponential* function $10^{\mathbb{x}}$ and the *integer logarithmic* function $\ell_{10}(\mathbb{x})$ [21]. The function $10^{\mathbb{x}}$ is defined for $\mathbb{x} \in \mathbb{N}$ in the normal sense. The function $\ell_{10}(\mathbb{x})$ is defined for positive integers $n \geq 1$: $\ell_{10}(n) = m$ if $10^m \leq n < 10^{m+1}$. Note that $10^{\mathbb{x}}$ is undefined for $\mathbb{x} < 0$ and $\ell_{10}(\mathbb{x})$ is undefined for $\mathbb{x} \leq 0$.

The syntax of ExpPA is obtained from that of PA by adding $10^{\mathbb{t}}$ and $\ell_{10}(\mathbb{t})$ to the definition of terms. An ExpPA term, denoted by $\mathbb{t}$, is of the form

$$\mathbb{t} \mathrel{\hat{=}} c \mid \mathbb{x} \mid \mathbb{t} + \mathbb{t} \mid \mathbb{t} - \mathbb{t} \mid 10^{\mathbb{t}} \mid \ell_{10}(\mathbb{t}),$$

where $\mathbb{x}$ is an integer variable and $c \in \mathbb{Z}$ is a constant integer. In addition, we require that $10^{\mathbb{t}}$ and $\ell_{10}(\mathbb{t})$ terms are well-defined, which restricts the interpretations of $\mathbb{t}$ therein.

A formula of ExpPA, denoted by ϕ, is of the form

$$\phi \mathrel{\hat{=}} \mathbb{t} \ \odot \ \mathbb{t} \mid c \diamond \mathbb{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists \mathbb{x}.\ \phi \mid \forall \mathbb{x}.\ \phi,$$

where $\odot \in \{=, <, >, \leq, \geq\}, \diamond \in \{|, \nmid\}$.

The semantics of ExpPA are defined similarly to that of PA with the only difference that partial functions are required to be well-defined. The quantifier-free and existential ExpPA formulas are defined similarly as well. We also assume that quantifier-free ExpPA formulas are free of negations, for the same reason we have explained for PA.

For convenience, when working with exponential and logarithmic functions, we denote $\lambda_{10}(\mathbb{t}) = 10^{\ell_{10}(\mathbb{t})}$. It is easy to show that for all $n \geq 1$, $\lambda_{10}(n) \leq n$ and $n < 10\lambda_{10}(n)$ holds.

Example 3 $10^{10^{\mathbb{x}}} + 10^{\mathbb{x}+\mathbb{y}} + 3\mathbb{y} \leq 2\mathbb{z} + 1$ is an ExpPA formula.

3.3 Flattening $\mathsf{STR}_{\mathsf{parseInt}}$ into ExpPA

We first recall the flattening approach for string constraints in [17], then show how to extend it to deal with parseInt.

A *flat domain restriction* for a string constraint $\varphi \in \mathsf{STR}_{\mathsf{parseInt}}$ is a function $\mathcal{F}_\varphi$ that maps each string variable $x \in \mathsf{SVar}(\varphi)$ to a tuple $(\mathfrak{L}_x, \psi_x)$, where $\mathfrak{L}_x$ is a

PFL over character variables V and ψ_x is a PA formula over V. In the following, we slightly abuse the notation $(\Sigma_{\text{num}})_\epsilon$ to consider it as a subset of $\mathbb{Z}$, by sending '0'$\to 0$, '1'$\to 1\ldots$, and $\epsilon \to 10$. By doing so, we can treat ψ_x as constraints on interpretations from V to $(\Sigma_{\text{num}})_\epsilon$. The intention of $\mathcal{F}_\varphi$ is to first translate the PFL $\mathfrak{L}_x$ into FLs via interpretations satisfying ψ_x, and then restrict the domain of x to those (finitely many) feasible FLs. Formally, the flattened semantics of $\varphi \in \mathsf{STR}_{\mathsf{parseInt}}$ is defined as $[\![\varphi]\!]_{\mathcal{F}_\varphi} = \{(I, I_p) \mid I \in [\![\varphi]\!], I_p \in [\![\psi_x]\!] \text{ and } I(x) \in I_p(\mathfrak{L}_x) \text{ for all } x \in \mathsf{SVar}(\varphi)\}$.

Under a flat domain restriction $\mathcal{F}_\varphi$, the *flattening* of $\varphi \in \mathsf{STR}_{\mathsf{parseInt}}$ is an arithmetic formula, denoted by $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$, that encodes the flattened semantics $[\![\varphi]\!]_{\mathcal{F}_\varphi}$. Theorem 1 below shows how to construct such $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$, here we briefly explain its idea. When the domain of a string variable $x \in \mathsf{SVar}(\varphi)$ is restricted to a FL $L_x = v_{x,1}(w_{x,1})^* v_{x,2} \cdots (w_{x,k_x})^* v_{x,k_x+1}$, we can express x as

$$x = v_{x,1}(w_{x,1})^{l_{x,1}} v_{x,2} \cdots (w_{x,k_x})^{l_{x,k_x}} v_{x,k_x+1}$$

where $l_{x,i}$ are fresh (non-negative) integer variables, called *flattening variables*, representing the number of iterations of $w_{x,i}$. Let us denote $\mathsf{PVar}_{\mathcal{F}_\varphi}(x) = \{l_{x,i} \mid i \in [k_x]\}$ and $\mathsf{PVar}_{\mathcal{F}_\varphi}(\varphi) = \bigcup_{x \in \mathsf{SVar}(\varphi)} \mathsf{PVar}_{\mathcal{F}_\varphi}(x)$ the set of introduced flattening variables for φ. Concretely speaking, $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is a formula over the integer variables $\mathsf{IVar}(\varphi)$ and flattening variables $\mathsf{PVar}_{\mathcal{F}_\varphi}(\varphi)$ (plus some other auxiliary variables), such that the interpretations $I_e : \mathsf{IVar}(\varphi) \cup \mathsf{PVar}_{\mathcal{F}_\varphi}(\varphi) \to \mathbb{Z}$ in $[\![\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)]\!]$ and the interpretations I in $[\![\varphi]\!]_{\mathcal{F}_\varphi}$ have the following correspondence:

$$I(x) = v_{x,1} w_{x,1}^{I_e(l_{x,1})} v_{x,2} \ldots w_{x,k_x}^{I_e(l_{x,k_x})} v_{x,k_x+1} \quad \text{for } x \in \mathsf{SVar}(\varphi)$$
$$I(\mathbb{x}) = I_e(\mathbb{x}) \quad \text{for } \mathbb{x} \in \mathsf{IVar}(\varphi).$$

Theorem 1 *Under the flat domain restriction $\mathcal{F}_\varphi$, the satisfiability of a $\mathsf{STR}_{\mathsf{parseInt}}$ formula φ can be reduced to the satisfiability of an existential* ExpPA *formula $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ in exponential time.*

Proof We aim to show that $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is a ExpPA formula. The formula $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is constructed inductively on the structure of φ:

$$\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi_1 \mathbin{\mathfrak{o}} \varphi_2) = \mathit{flatten}_{\mathcal{F}_\varphi}(\varphi_1) \mathbin{\mathfrak{o}} \mathit{flatten}_{\mathcal{F}_\varphi}(\varphi_2)$$
$$\mathit{flatten}_{\mathcal{F}_\varphi}(\neg\varphi_1) = \neg \mathit{flatten}_{\mathcal{F}_\varphi}(\varphi_1)$$

where $\mathfrak{o} \in \{\wedge, \vee\}$. Therefore, it is sufficient to show how to construct $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ for atomic constraints φ. When φ is a string equality constraint $t_1 = t_2$ or a regular constraint $t \in \mathcal{A}$, since $\mathsf{parseInt}$ will not occur in φ, the construction of $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is essentially the same as that in [17, Section 7], thus omitted here. In this case, $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is a PA formula and can be computed in polynomial time. In the following, we show how to construct $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathbb{t}_1 \odot \mathbb{t}_2)$ where $\mathsf{parseInt}(t)$ may occur in $\mathbb{t}_1$ or $\mathbb{t}_2$.

Note that the flat domain restriction $\mathcal{F}_\varphi$ maps a string variable x to $(\mathfrak{L}_x, \psi_x)$, since Σ_{num} is a finite alphabet, the number of FLs corresponding to the PFL $\mathfrak{L}_x$ is also finite. Therefore, we can enumerate all flat languages $\{L_x\}$ corresponding to $\mathfrak{L}_x$, then check that if the string

constraint φ and the interpretation restriction ψ_x are both satisfiable with $x \in L_x$ for each L_x. In the rest of the proof, we will assume that, for each $x \in \mathsf{SVar}(\varphi)$, x belongs to a flat language $L_x = v_{x,1}(w_{x,1})^* v_{x,2} \cdots (w_{x,k_x})^* v_{x,k_x+1}$ with $v_{x,\text{-}}, w_{x,\text{-}} \in \Sigma^*_{num}$ instead of a parametric flat language $\mathfrak{L}_x$.

For simplicity, we assume that each occurrence of $\mathsf{parseInt}$ (resp. $\mathsf{len}(t)$) in $\mathbb{t}_1 \odot \mathbb{t}_2$ is of the form $\mathsf{parseInt}(x)$ (resp. $\mathsf{len}(x)$) for a string variable x. Otherwise, we introduce a fresh variable x' to replace t in $\mathsf{parseInt}(t)$ or $\mathsf{len}(t)$ and add the constraint $x' = t$. Then $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathbb{t}_1 \odot \mathbb{t}_2)$ is obtained from $\mathbb{t}_1 \odot \mathbb{t}_2$ by replacing $\mathsf{parseInt}(x)$ with $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathsf{parseInt}(x))$ and $\mathsf{len}(x)$ with $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathsf{len}(x))$, where

- $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathsf{parseInt}(x)) \hat{=} \mathbb{t}_{x,1}$ such that $(\mathbb{t}_{x,i})_{i\in[k_x+1]}$ are inductively defined as follows:
  - for $i = k_x + 1$,
  $$\mathbb{t}_{x,i} = \mathsf{parseInt}(v_{x,k_x+1})$$
  - for $i \in [k_x]$,
  $$\mathbb{t}_{x,i} = \mathsf{parseInt}(v_{x,i})10^{l_i^{\text{suf}}+\mathsf{len}(w_{x,i})l_{x,i}} + \mathsf{parseInt}(w_{x_i})\frac{(10^{\mathsf{len}(w_{x,i})l_{x,i}}-1)}{(10^{\mathsf{len}(w_{x,i})}-1)}10^{l_i^{\text{suf}}} + \mathbb{t}_{x,i+1}$$
  where $l_i^{\text{suf}} = \sum_{i+1\le j\le k_x} \mathsf{len}(w_{x,j})l_{x,j} + \sum_{i+1\le j\le k_x+1} \mathsf{len}(v_{x,j})$.
- $\mathit{flatten}_{\mathcal{F}_\varphi}(\mathsf{len}(x)) \hat{=} \sum_{i\in[k_x]} \mathsf{len}(w_{x,i})l_{x,i} + \sum_{i\in[k_x+1]} \mathsf{len}(v_{x,i})$.

As a result, we eliminate all occurrence of string variables in φ by introducing flattening variables. The obtained formula $\mathit{flatten}_{\mathcal{F}_\varphi}(\varphi)$ is an existential ExpPA formula. Since the enumeration of all possible FLs requires exponential time, while other operations require only polynomial time, the whole procedure can be carried out in exponential time. Therefore the theorem is proved.

□

Later in this paper, we will prove that the decision problem for existential ExpPA formulas is in 3-EXPTIME (see Theorem 10 in Section 5). Utilizing this fact, we have

Theorem 2 *The satisfiability problem of* $\mathsf{STR}_{\mathsf{parseInt}}$ *under flat domain restrictions is in 3-EXPTIME.*

The following example helps to illustrate the main idea of the flattening technique.

Example 4 Suppose $\mathsf{parseInt}(x) = 2\mathbb{x}$ is an atomic constraint and $\mathcal{F}_\varphi(x) = (1^*2^*, \mathsf{true})$. Then

$$\mathit{flatten}_{\mathcal{F}_\varphi}(\mathsf{parseInt}(x) = 2\mathbb{x}) \hat{=} 1 \cdot \frac{10^{l_{x,1}}-1}{10-1} \cdot 10^{l_{x,2}} + 2 \cdot \frac{10^{l_{x,2}}-1}{10-1} = 2\mathbb{x}$$

$$\equiv 10^{l_{x,1}+l_{x,2}} - 10^{l_{x,2}} + 2 \cdot 10^{2l_{x,2}} - 2 = 18\mathbb{x}$$
$$\equiv 10^{l_{x,1}+l_{x,2}} + 10^{l_{x,2}} = 18\mathbb{x} + 2.$$

Remark 1 In [17, Section 8], the authors have already informally described the ExpPA encoding for the parseInt functions. However, they only pointed out the difficulties brought by exponential functions without providing a decision procedure. The algorithm of [17] allowed only a very restricted form of flat domain restrictions for parseInt constraints. Namely, for a term parseInt(x), $\mathfrak{L}_x$ should be of the form $(\mathfrak{a}_1)^*\mathfrak{b}_1 \dots \mathfrak{b}_m$ where $\mathfrak{a}_1$ must be interpreted to 0. Consequently, the treatment of general flat domain restrictions has been left as an open problem. In this paper, we present a complete solution to this problem.

4 Quantifier elimination procedure for ExpPA

Semënov first proved that ExpPA admits quantifier elimination in [20], thus its satisfiability problem is decidable; however, he did not give a concrete quantifier elimination procedure. Later in [21], Cherlin and Point proposed the first quantifier elimination procedure for ExpPA, which is referred to as Cherlin and Point's procedure in this paper. In [22], Point revisited this problem and rewrite the procedure for better clarity.

In this section, we first present our new quantifier elimination procedure for ExpPA, which is improved from Cherlin and Point's procedure. Then, we discuss the differences between the two procedures. Throughout this section, we will always assume that formulas are in PNF.

Theorem 3 ([21, Prop. 1]) ExpPA *admits quantifier elimination.*

As $\forall\mathbb{x}.\ \varphi$ is equivalent to $\neg\exists\mathbb{x}.\ \neg\varphi$, to prove Theorem 3, we only need to show that every existentially quantified ExpPA formula $\exists\mathbb{x}.\ \varphi \in$ ExpPA, where φ is quantifier-free, can be transformed into an equivalent quantifier-free formula. We use the abbreviation $\exists\mathbb{x}(\in\mathbb{N}).\varphi$ for $\exists\mathbb{x}.\varphi \wedge \mathbb{x} \geq 0$ to stress that the quantified variable $\mathbb{x}$ is non-negative. A sketch of our procedure is presented in Algorithm 1.

4.1 Normalization

Given an ExpPA formula $\exists\mathbb{x}.\varphi$ with φ quantifier-free, in general, the structure of φ can be quite complicated due to nested applications of exponential and integer logarithm functions. The normalization step rewrites φ into a simpler but equivalent form so that there exist only two kinds of atomic formulas: inequality atomic formulas of the form

$$\sum_{j=1}^{n} a_j 10^{\mathbb{x}_j} + \sum_{k=1}^{n} b_k \mathbb{x}_k \leq \mathbb{t}(\vec{\mathbb{y}}) \tag{1}$$

Algorithm 1: ExpPA quantifier elimination procedure

Input : ExpPA formula $\exists \mathbb{x}.\varphi$ with φ quantifier-free.

Output: an equivalent quantifier-free formula

$\varphi' \leftarrow$ normalize φ with respect to $\mathbb{x}$ (Sect. 4.1)

▷ $\exists \vec{\mathbb{x}} (\in \mathbb{N}^n).\varphi'$ is equivalent with $\exists \mathbb{x}.\varphi$

$\mathcal{S}_n \leftarrow$ enumerate linear orders among $\mathbb{x}_1, \cdots, \mathbb{x}_n$ (Sect. 4.2)

foreach $\sigma \in \mathcal{S}_n$ **do**

  $\varphi'_\sigma \leftarrow \varphi' \wedge \bigwedge_{i \in [n-1]} \mathbb{x}_{\sigma(i)} \leq \mathbb{x}_{\sigma(i+1)}$

  $i \leftarrow n$

  while $i > 0$ **do**

    eliminate exponential occurrences of $\mathbb{x}_{\sigma(i)}$ in φ'_σ (Sect. 4.3)

    eliminate linear occurrences of $\mathbb{x}_{\sigma(i)}$ in φ'_σ (Cooper's algorithm [23])

    ▷ variable $\mathbb{x}_{\sigma(i)}$ is removed

    $i \leftarrow i - 1$

  end

  ▷ all quantified variables in φ' are removed

  $\varphi''_\sigma \leftarrow \varphi'_\sigma$

end

return $\bigvee_{\sigma \in \mathcal{S}_n} \varphi''_\sigma$

and (in)divisibility atomic formulas of the form

$$d \diamond \Big(\sum_{j=1}^{n} a_j 10^{\mathbb{x}_j} + \sum_{k=1}^{n} b_k \mathbb{x}_k + \mathbb{t}(\vec{\mathbb{y}}) \Big), \quad \diamond \in \{|, \nmid\} \tag{2}$$

where $\{\mathbb{x}_i\}_{i \in [n]}$ represent the introduced (existentially quantified) variables during rewriting, $\vec{\mathbb{y}}$ is the set of free variables and $\mathbb{t}(\vec{\mathbb{y}})$ collects all terms not containing $\mathbb{x}_i$.

Let us say a term $\mathbb{t}$ *properly contains* a variable $\mathbb{x}$ if $\mathbb{x}$ occurs in $\mathbb{t}$ and $\mathbb{t} \neq \mathbb{x}$. Formally speaking, we would like to rewrite $\exists \mathbb{x}.\varphi$ into an equivalent formula $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_n.\varphi'$ such that: 1) φ' contains no occurrences of $\ell_{10}(\mathbb{t})$ where $\mathbb{t}$ contains some $\mathbb{x}_i$; 2) φ' contains no occurrences of $10^{\mathbb{t}}$ where $\mathbb{t}$ properly contains some $\mathbb{x}_i$; 3) all atomic formulas are of the form Eq. (1) or Eq. (2); 4) all quantified variables $\{\mathbb{x}_i\}_{i \in [n]}$ are non-negative. We say a formula is *normalized* if it satisfies these constraints.

The normalization step comprises following four sub-steps that guarantee the above four requirements respectively. For consistency and simplicity, we first rename the original quantified variable $\mathbb{x}$ as $\mathbb{x}_1$; when a fresh variable $\mathbb{x}_j$ is introduced, we assume $j \in \mathbb{Z}^+$ is the smallest positive integer such that $\mathbb{x}_j$ has not been used. We now describe the four sub-steps of normalization:

(1) Encode logarithm functions. For each occurrence of $\ell_{10}(\mathbb{t})$ in φ such that $\mathbb{t}$ contains some $\mathbb{x}_i$, introduce a fresh variable $\mathbb{x}_j$ and replace all occurrences of $\ell_{10}(\mathbb{t})$ by $\mathbb{x}_j$, moreover, add the constraint $10^{\mathbb{x}_j} \leq \mathbb{t} < 10 \cdot 10^{\mathbb{x}_j}$ as a conjunct. Let the resulting formula be φ_1, then φ_1 contains no $\ell_{10}(\mathbb{t})$ terms with $\mathbb{x}_i$ in $\mathbb{t}$.

(2) Flatten exponential terms. For each occurrence of the $10^{\mathbb{t}}$ in φ_1 such that $\mathbb{t}$ properly contains some $\mathbb{x}_i$, introduce a fresh variable $\mathbb{x}_j$ and replace all occurrences of $10^{\mathbb{t}}$ by $10^{\mathbb{x}_j}$, moreover, add the constraint $\mathbb{x}_j = \mathbb{t}$ as a conjunct. Let φ_2 denote the resulting formula, then φ_2 has no $10^{\mathbb{t}}$ terms in which $\mathbb{t}$ properly contains some $\mathbb{x}_i$.

(3) Transform atomic formulas. Replace every occurrence of $\mathbb{t}_1 \geq \mathbb{t}_2$ with $\mathbb{t}_2 \leq \mathbb{t}_1$; replace every occurrence of $\mathbb{t}_1 < \mathbb{t}_2$ (resp. $\mathbb{t}_1 > \mathbb{t}_2$) with $\mathbb{t}_1 \leq \mathbb{t}_2 - 1$ (resp. $\mathbb{t}_2 \leq \mathbb{t}_1 - 1$); replace every occurrence of $\mathbb{t}_1 = \mathbb{t}_2$ with $\mathbb{t}_2 \leq \mathbb{t}_1 \wedge \mathbb{t}_1 \leq \mathbb{t}_2$. Let φ_3 denote the resulting formula, then all atomic formulas in φ_3 are of the form Eq. (1) or Eq. (2).

(4) Rename variables. Note that all introduced variables $\mathbb{x}_j$ with $j \neq 1$ in substep (1) and (2) should be non-negative due to the existence of $10^{\mathbb{x}_j}$ terms. For consistency, we also want $\mathbb{x}_1$ to be non-negative, so we do as follows: if $10^{\mathbb{x}_1}$ does not appear in φ_3, we introduce a fresh variable $\mathbb{x}_j$ and substitute all occurrences of $\mathbb{x}_1$ by $\mathbb{x}_1 - \mathbb{x}_j$; otherwise $\mathbb{x}_1 \geq 0$ can be guaranteed. Let φ' denote the resulting formula.

Let $\vec{\mathbb{x}} = \mathbb{x}_1, \ldots, \mathbb{x}_n$ be an enumeration of introduced variables together with original $\mathbb{x}_1$, the result of the normalization procedure is $\exists \vec{\mathbb{x}} (\in \mathbb{N}^n).\ \varphi'$, which is equivalent to $\exists \mathbb{x}.\varphi$. We use the following example to illustrate the whole procedure.

Example 5 Consider $\exists \mathbb{x}_1.\varphi(\mathbb{x}_1, \mathbb{y})$ with

$$\varphi \mathrel{\hat{=}} 3\mathbb{y} \leq \ell_{10}(9 \cdot 10^{2\mathbb{x}_1} + 200\mathbb{x}_1).$$

In step (1), since $\ell_{10}(9 \cdot 10^{2\mathbb{x}_1} + 200\mathbb{x}_1)$ contains $\mathbb{x}_1$, we replace this term by a fresh variable $\mathbb{x}_2$:

$$\varphi_1 \mathrel{\hat{=}} 3\mathbb{y} \leq \mathbb{x}_2 \wedge 10^{\mathbb{x}_2} \leq 9 \cdot 10^{2\mathbb{x}_1} + 200\mathbb{x}_1 < 10 \cdot 10^{\mathbb{x}_2}.$$

In step (2), note that the exponent $2\mathbb{x}_1$ properly contains $\mathbb{x}_1$, we replace $2\mathbb{x}_1$ by a fresh variable $\mathbb{x}_3$:

$$\varphi_2 \mathrel{\hat{=}} 3\mathbb{y} \leq \mathbb{x}_2 \wedge 10^{\mathbb{x}_2} \leq 9 \cdot 10^{\mathbb{x}_3} + 200\mathbb{x}_1 < 10 \cdot 10^{\mathbb{x}_2} \wedge \mathbb{x}_3 = 2\mathbb{x}_1.$$

In step (3), we rewrite all (in)equalities into the form $\mathbb{t}_1 \leq \mathbb{t}_2$:

$$\begin{aligned}\varphi_3 \mathrel{\hat{=}} 3\mathbb{y} \leq \mathbb{x}_2 \wedge 10^{\mathbb{x}_2} \leq 9 \cdot 10^{\mathbb{x}_3} + 200\mathbb{x}_1 \wedge 9 \cdot 10^{\mathbb{x}_3} + 200\mathbb{x}_1 \leq 10 \cdot 10^{\mathbb{x}_2} - 1 \\ \wedge \mathbb{x}_3 \leq 2\mathbb{x}_1 \wedge 2\mathbb{x}_1 \leq \mathbb{x}_3.\end{aligned}$$

Finally, in step (4), since $\mathbb{x}_1$ does not occur as an exponent in φ_3, we replace $\mathbb{x}_1$ by $\mathbb{x}_1 - \mathbb{x}_4$ with $\mathbb{x}_4$ a fresh variable:

$$\begin{aligned}\varphi' \mathrel{\hat{=}} 3\mathbb{y} \leq \mathbb{x}_2 \wedge 10^{\mathbb{x}_2} \leq 9 \cdot 10^{\mathbb{x}_3} + 200(\mathbb{x}_1 - \mathbb{x}_4) \wedge 9 \cdot 10^{\mathbb{x}_3} + 200(\mathbb{x}_1 - \mathbb{x}_4) \\ \leq 10 \cdot 10^{\mathbb{x}_2} - 1 \wedge \mathbb{x}_3 \leq 2(\mathbb{x}_1 - \mathbb{x}_4) \wedge 2(\mathbb{x}_1 - \mathbb{x}_4) \leq \mathbb{x}_3.\end{aligned}$$

After separating terms containing $\mathbb{x}_i$ and $\mathbb{y}$, all atomic formulas are of the form Eq. (1).

4.2 Enumerating all variable orders

After normalization, the formula becomes $\exists \vec{\mathbb{x}} (\in \mathbb{N}^n).\ \varphi'$. Next, we enumerate all linear orders of $\{\mathbb{x}_1, \ldots, \mathbb{x}_n\}$ and represent each linear order by a permutation $\sigma \in \mathcal{S}_n$ (where $\mathcal{S}_n$ is the permutation group on $[n]$), with the intention that $\mathbb{x}_{\sigma(n)} \geq \cdots \geq \mathbb{x}_{\sigma(1)} (\geq 0)$.

Assuming a linear order $\sigma \in \mathcal{S}_n$ of $\{\mathbb{x}_1, \ldots, \mathbb{x}_n\}$, we then consider $\varphi'_\sigma = \exists \vec{\mathbb{x}}.\ \varphi' \wedge \bigwedge_{i\in[n-1]} \mathbb{x}_{\sigma(i)} \leq \mathbb{x}_{\sigma(i+1)}$ and eliminate the quantifiers one by one and from $\exists \mathbb{x}_{\sigma(n)}$ to $\exists \mathbb{x}_{\sigma(1)}$. To eliminate the quantifier of $\mathbb{x}_{\sigma(i)}$, we first eliminate all exponential occurrences of $\mathbb{x}_{\sigma(i)}$, i.e., $10^{\mathbb{x}_{\sigma(i)}}$, using the fact that $\mathbb{x}_{\sigma(i)}$ is the largest among the remaining quantified variables (see Lemma 4); linear occurrences are eliminated further, by applying Cooper's algorithm [23]. Let φ''_σ denote the resulting formula.

Finally, after repeating this procedure for all linear orders, $\exists \vec{\mathbb{x}}.\ \varphi'$ is transformed into the quantifier-free formula $\bigvee_{\sigma\in\mathcal{S}_n} \varphi''_\sigma$.

4.3 Elimination of exponential occurrences of variables

For $i \in [n]$, let $\exists \mathbb{x}_{\sigma(1)} \ldots \exists \mathbb{x}_{\sigma(i)}.\ \varphi''_{\sigma,i}(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ be the formula obtained from φ'_σ by eliminating the quantifiers $\exists \mathbb{x}_{\sigma(n)}$, …, $\exists \mathbb{x}_{\sigma(i+1)}$. In what follows, we show how to eliminate the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ in $\varphi''_{\sigma,i}$. The elimination is *local* in the sense that it is applied to the atomic formulas independently.

Recall that after normalization, the atomic formulas are of the form Eq. (1) or Eq. (2), the two cases are treated differently in the following.

4.3.1 Inequality atomic formulas

In the following, we illustrate how to eliminate the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ from inequality atomic formulas of the form

$$\begin{aligned} &\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}}) \hat{=} \\ &a_i 10^{\mathbb{x}_{\sigma(i)}} + \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} \leq \mathbb{t}(\vec{\mathbb{y}}) \end{aligned} \tag{3}$$

where $a_i \neq 0$, and $\mathbb{t}(\vec{\mathbb{y}})$ is the sum of all other terms without $\mathbb{x}_{\sigma(1)}, \ldots, \mathbb{x}_{\sigma(i)}$. The elimination of the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ in $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ relies on the following lemma.

Lemma 4 *Let $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ be of form Eq. (3) with $a_i \neq 0$. Let $A \hat{=} 1 + \sum_{j=1}^{i-1} |a_j|$, $B \hat{=} 1 + \sum_{j=1}^{i} |b_j|$, $\delta \hat{=} \ell_{10}(A) + 2$, and $\gamma \hat{=} 2\ell_{10}(B) + 3$. For a given interpretation I such that $I \models \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta$, we have*

- *for $a_i > 0$, let $\alpha(\vec{\mathbb{y}}) \hat{=} \ell_{10}(\mathbb{t}(\mathbb{y})) - \ell_{10}(a_i)$, then*
  - *if $\alpha(\vec{\mathbb{y}})$ is undefined, i.e., $I \models \mathbb{t}(\vec{\mathbb{y}}) \leq 0$, then $I \not\models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$;*
  - *if $I \models \mathbb{x}_{\sigma(i)} \geq \alpha(\vec{\mathbb{y}}) + 2$, then $I \not\models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$;*
  - *if $I \models \mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1$, then $I \models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$.*
- *for $a_i < 0$, let $\alpha(\vec{\mathbb{y}}) \hat{=} \ell_{10}(-\mathbb{t}(\mathbb{y})) - \ell_{10}(-a_i)$, then*
  - *if $\alpha(\vec{\mathbb{y}})$ is undefined, i.e., $I \models \mathbb{t}(\vec{\mathbb{y}}) \leq 0$, then $I \models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$;*
  - *if $I \models \mathbb{x}_{\sigma(i)} \geq \alpha(\vec{\mathbb{y}}) + 2$, then $I \models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$;*
  - *if $I \models \mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1$, then $I \not\models \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$.,*

Intuitively, the lemma states that, when $a_i 10^{\mathbb{x}_{\sigma(i)}}$ dominates the left-hand-side of $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ and $\mathbb{t}(\vec{\mathbb{y}}) > 0$, if either $\mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1$ or $\mathbb{x}_{\sigma(i)} \geq \alpha(\vec{\mathbb{y}}) + 2$ holds, the truth value of $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ can be determined directly. The proof of Lemma 4 is based on the following Proposition 5 and Proposition 6.

Proposition 5 *Given $n \geq 1$, if $\mathbb{x} \geq 2\ell_{10}(n) + 1$, then $n\mathbb{x} \leq 10^{\mathbb{x}}$ holds.*

Proof For $1 \leq n \leq 9$, i.e., $\ell_{10}(n) = 0$, we have $n\mathbb{x} \leq 10\mathbb{x} \leq 10^{\mathbb{x}}$ for all $\mathbb{x} \in \mathbb{N}$.
For $n \geq 10$, i.e., $\ell_{10}(n) \geq 1$, then we have

$$\begin{aligned}
10^{\mathbb{x}} &= 10^{\ell_{10}(n)+1} 10^{\mathbb{x}-\ell_{10}(n)-1} \\
&\geq 10^{\ell_{10}(n)+1} 10(\mathbb{x} - \ell_{10}(n) - 1) \\
&\geq 10^{\ell_{10}(n)+1} \big((5\mathbb{x} - 10\ell_{10}(n) - 5) + (5\mathbb{x} - 5)\big) \\
&\geq 10^{\ell_{10}(n)+1} (5\mathbb{x} - 5) && \text{(by } \mathbb{x} \geq 2\ell_{10}(n) + 1\text{)} \\
&\geq 10^{\ell_{10}(n)+1} \big(\mathbb{x} + (4\mathbb{x} - 5)\big) \\
&\geq 10^{\ell_{10}(n)+1} \mathbb{x} && \text{(by } \mathbb{x} \geq 2\ell_{10}(n) + 1 \geq 3\text{)} \\
&\geq n\mathbb{x}
\end{aligned}$$

□

Proposition 6 can be seen as a special case of Lemma 4 where the left-hand-side of $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ contains only one term $a_i 10^{\mathbb{x}_{\sigma(i)}}$. In the following, we allow constants in ExpPA formulas to be fractions. For instance, $\mathbb{x} + \frac{1}{2} \leq \mathbb{y}$ can be understood as $2\mathbb{x} + 1 \leq 2\mathbb{y}$.

Proposition 6 *Given $a > 0$, if $\mathbb{x} \leq \ell_{10}(\mathbb{y}) - \ell_{10}(a) - 1$, then $(a + \frac{1}{2}) \cdot 10^{\mathbb{x}} \leq \mathbb{y}$; if $\mathbb{x} \geq \ell_{10}(\mathbb{y}) - \ell_{10}(a) + 2$, then $(a - \frac{1}{2}) \cdot 10^{\mathbb{x}} \geq \mathbb{y}$.*

Proof Recall that $\lambda_{10}(\mathbb{t}) = 10^{\ell_{10}(\mathbb{t})}$ and $\lambda_{10}(n) \leq n < 10\lambda_{10}(n)$ for all $n \geq 1$. If $\mathbb{x} \leq \ell_{10}(\mathbb{y}) - \ell_{10}(a) - 1$, we have

$$(a + \frac{1}{2}) \cdot 10^{\mathbb{x}} \leq (a + \frac{1}{2}) \cdot \frac{\lambda_{10}(\mathbb{y})}{10\lambda_{10}(a)} = \frac{a + \frac{1}{2}}{10\lambda_{10}(a)} \lambda_{10}(\mathbb{y}) \leq \lambda_{10}(\mathbb{y}) \leq \mathbb{y}.$$

Now, suppose that $\mathbb{x} \geq \ell_{10}(\mathbb{y}) - \ell_{10}(a) + 2$,

$$(a - \frac{1}{2}) \cdot 10^{\mathbb{x}} \geq (a - \frac{1}{2}) \frac{100\lambda_{10}(\mathbb{y})}{\lambda_{10}(a)} = \frac{10(a - \frac{1}{2})}{\lambda_{10}(a)} 10\lambda_{10}(\mathbb{y}) \geq 10\lambda_{10}(\mathbb{y}) \geq \mathbb{y}$$

□

Proof of Lemma 4 We only prove for the case $a_i > 0$, the $a_i < 0$ case is symmetric. The proof is twofold, given an interpretation $I \models \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta$:

1) First, we prove that the left-hand-side of $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ is dominated by $a_i 10^{\mathbb{x}_{\sigma(i)}}$, i.e. ,

$$(a_i - \frac{1}{2})10^{\mathbb{x}_{\sigma(i)}} < a_i 10^{\mathbb{x}_{\sigma(i)}} + \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} < (a_i + \frac{1}{2})10^{\mathbb{x}_{\sigma(i)}} \quad (4)$$

To obtain the formula above, we need to bound the absolute values $|\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}}|$ and $|\sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)}|$ by some fraction of $10^{\mathbb{x}_{\sigma(i)}}$ respectively. Here we choose the bound to be $\frac{1}{10}10^{\mathbb{x}_{\sigma(i)}}$.

Using $\mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta$ and the linear order $\mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} \geq \cdots \geq \mathbb{x}_{\sigma(1)}$, we can prove that

$$\begin{aligned} |\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}}| &\leq \sum_{j=1}^{i-1} |a_j| 10^{\mathbb{x}_{\sigma(j)}} \\ &< A \cdot 10^{\mathbb{x}_{\sigma(i)} - \delta} \\ &\leq \frac{A}{100\lambda_{10}(A)} 10^{\mathbb{x}_{\sigma(i)}} \\ &< \frac{1}{10} 10^{\mathbb{x}_{\sigma(i)}}. \qquad \text{(by } \mathbb{x} < 10\lambda_{10}(\mathbb{x})\text{)} \end{aligned}$$

Similarly, using $\mathbb{x}_{\sigma(i)} \geq \gamma$, we have

$$\begin{aligned} |\sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)}| &\leq \sum_{k=1}^{i} |b_k| \mathbb{x}_{\sigma(k)} \\ &< B \cdot \mathbb{x}_{\sigma(i)} \\ &= \frac{1}{10} \cdot 10B \cdot \mathbb{x}_{\sigma(i)} \\ &\leq \frac{1}{10} 10^{\mathbb{x}_{\sigma(i)}}. \qquad \text{(by setting } n = 10B \text{ in Prop. 5)} \end{aligned}$$

Combining the two inequalities above, we have

$$|\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)}| < \frac{3}{10} 10^{\mathbb{x}_{\sigma(i)}} < \frac{1}{2} 10^{\mathbb{x}_{\sigma(i)}},$$

from which Eq. (4) can be obtained.

2) Then, we give the sufficient conditions for $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ to hold or not by comparing $\mathbb{t}(\vec{\mathbb{y}})$ with $(a_i \pm \frac{1}{2})10^{\mathbb{x}_{\sigma(i)}}$. Note that when $\mathbb{t}(\vec{\mathbb{y}}) \leq 0$, $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ does not hold since $0 < (a_i - \frac{1}{2})10^{\mathbb{x}_{\sigma(i)}}$, so we only need to consider the $\mathbb{t}(\vec{\mathbb{y}}) > 0$ case.

Utilizing Proposition 6, we obtain that

$$\begin{aligned} \mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1 &\implies (a_i + \frac{1}{2}) \cdot 10^{\mathbb{x}_{\sigma(i)}} \leq \mathbb{t}(\vec{\mathbb{y}}) \\ &\implies \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}}) \text{ is satisfied} \end{aligned} \quad (5)$$

and

$$\begin{aligned} \mathbb{x}_{\sigma(i)} \geq \alpha(\vec{\mathbb{y}}) + 2 &\implies (a_i - \frac{1}{2}) \cdot 10^{\mathbb{x}_{\sigma(i)}} \geq \mathbb{t}(\vec{\mathbb{y}}) \\ &\implies \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}}) \text{ is unsatisfied} \end{aligned} \quad (6)$$

Thus, the lemma is proved. □

For $a_i > 0$, by Lemma 4, $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ (for readability, denoted by τ below) can be replaced by an equivalent formula without exponential occurrences of $\mathbb{x}_{\sigma(i)}$:

$$\begin{aligned}
\tau \leftrightarrow \quad & (\tau \wedge \mathbb{x}_{\sigma(i)} < \gamma) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} < \mathbb{x}_{\sigma(i-1)} + \delta) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta) \\
\leftrightarrow \quad & (\tau \wedge \mathbb{x}_{\sigma(i)} < \gamma) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} < \mathbb{x}_{\sigma(i-1)} + \delta) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \wedge \mathbb{t}(\vec{\mathbb{y}}) \leq 0) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \wedge \mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \wedge \alpha(\vec{\mathbb{y}}) - 1 < \mathbb{x}_{\sigma(i)} < \alpha(\vec{\mathbb{y}}) + 2) \\
& \vee (\tau \wedge \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \wedge \mathbb{x}_{\sigma(i)} \geq \alpha(\vec{\mathbb{y}}) + 2).
\end{aligned}$$

Finally, we have:

$$\begin{aligned}
\tau \leftrightarrow \quad & \bigvee_{p=0}^{\gamma-1} (\mathbb{x}_{\sigma(i)} = p \wedge \tau[p/\mathbb{x}_{\sigma(i)}]) \\
& \vee \bigvee_{p=0}^{\delta-1} (\mathbb{x}_{\sigma(i)} = \mathbb{x}_{\sigma(i-1)} + p \wedge \tau[\mathbb{x}_{\sigma(i-1)} + p/\mathbb{x}_{\sigma(i)}]) \\
& \vee \left(\mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \wedge \mathbb{x}_{\sigma(i)} \leq \alpha(\vec{\mathbb{y}}) - 1 \right) \\
& \vee \bigvee_{p=0,1} \begin{pmatrix} \mathbb{x}_{\sigma(i)} \geq \gamma \wedge \mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta \\ \wedge \mathbb{x}_{\sigma(i)} = \alpha(\vec{\mathbb{y}}) + p \wedge \tau[\alpha(\vec{\mathbb{y}}) + p/\mathbb{x}_{\sigma(i)}] \end{pmatrix}
\end{aligned} \tag{7}$$

The elimination of the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ for the case $a_i < 0$ is similar.

We would like to make a few remarks about Lemma 4: 1) when $i = 1$, we remove the constraint $\mathbb{x}_{\sigma(i)} \geq \mathbb{x}_{\sigma(i-1)} + \delta$, 2) The constant 1 in the definition of A, B is to ensure that $\ell_{10}(A), \ell_{10}(B)$ are well-defined. Besides, the lemma still holds when the base of exponential functions is changed to any natural number $n \geq 2$ (constants in δ, γ should be changed accordingly).

4.3.2 Divisibility atomic formulas

For divisibility atomic formulas, we can enumerate $\mathbb{x}_{\sigma(i)}$ within a finite range thanks to the periodic property of divisibility relation. Consider a divisibility atomic formula

$$\begin{aligned}
& \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}}) \hat{=} \\
& d \mid \left(a_i 10^{\mathbb{x}_{\sigma(i)}} + \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} + \mathbb{t}(\vec{\mathbb{y}}) \right)
\end{aligned} \tag{8}$$

with $a_i \neq 0$ and $d \geq 1$. The indivisibility case can be treated analogously.

Let $d = 2^{r_1}5^{r_2}d_0$ such that d_0 is divisible by neither 2 nor 5. Denote $r = \max(r_1, r_2)$, we have $d|(10^r d_0)$. Since 10 and d_0 are relatively prime, according to Euler's theorem (cf. [24]), $10^{\phi(d_0)} \equiv 1 \bmod d_0$, where ϕ is the Euler function, i.e., $\phi(d_0)$ counts the positive integers up to d_0 that are relatively prime to d_0. Suppose $10^{\phi(d_0)} = kd_0 + 1$ for some $k \in \mathbb{N}$, we have, for every $n \in \mathbb{N}$ with $n \geq r$,

$$\begin{aligned} 10^{n+\phi(d_0)} &\equiv 10^{n-r}10^r(kd_0+1) \bmod d \\ &\equiv 10^{n-r}(k10^r d_0 + 10^r) \bmod d \\ &\equiv 10^{n-r}(0+10^r) \bmod d \\ &\equiv 10^n \bmod d. \end{aligned}$$

Therefore, $\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})$ is equivalent to the following formula:

$$\begin{aligned} &\bigvee_{p=0}^{r-1} \tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}, \vec{\mathbb{y}})[p/\mathbb{x}_{\sigma(i)}] \\ &\vee \left(\begin{array}{l} \mathbb{x}_{\sigma(i)} \geq r \wedge \\ \bigvee_{p=0}^{\phi(d_0)-1} \left(\begin{array}{l} \phi(d_0) \mid (\mathbb{x}_{\sigma(i)} - r - p) \wedge \\ d \mid \left(\begin{array}{l} a_i 10^{r+p} + \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \\ \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} + \mathbb{t}(\vec{\mathbb{y}}) \end{array} \right) \end{array} \right) \end{array} \right), \end{aligned} \tag{9}$$

where the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ disappear. When $\mathbb{x}_{\sigma(i)} > r$, we only need to enumerate $\mathbb{x}_{\sigma(i)}$ in one period, i.e., from r to $r + \phi(d_0) - 1$.

For a special case $d_0 = 1$, Eq. (9) can be simplified into the form

$$\bigvee_{p=0}^{r-1} \tau[p/\mathbb{x}_{\sigma(i)}] \vee \left(\mathbb{x}_{\sigma(i)} \geq r \wedge d \mid \left(\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} + \mathbb{t}(\vec{\mathbb{y}}) \right) \right).$$

4.4 Comparison with Cherlin and Point's procedure

In the following, we discuss the differences between our ExpPA quantifier elimination procedure (Algorithm 1) with Cherlin and Point's procedure described in [22] (essentially same as [21]), summarized in Algorithm 2.

Firstly, the underlying theory of [22] differs from ExpPA in two aspects: 1) the domain is limited to $\mathbb{N}$, instead of $\mathbb{Z}$; 2) the vocabulary includes integer division functions $\frac{\mathbb{t}}{d}(d \in \mathbb{Z}^+)$, instead of divisibility predicates $d \mid \mathbb{t}(d \in \mathbb{Z}^+)$. Despite these differences in definition, we can always translate a formula in the theory of [22] into an equivalent ExpPA formula. To achieve this, we change the domain of variables from $\mathbb{N}$ to $\mathbb{Z}$ and require variables to be non-negative; furthermore, we replace each term of the form $\frac{\mathbb{t}}{d}(d \in \mathbb{Z}^+)$ with a fresh variable $\mathbb{z}$ and add $d\mathbb{z} \leq \mathbb{t} < d\mathbb{z} + d$ as a conjunct. In other words, ExpPA is a strict superset of the theory in [22], when the base of exponential functions is fixed.

In the sequel, we refer to both divisibility predicates and integer division functions as *divisibility constraints*. The overall frameworks of Algorithm 1 and

Algorithm 2: Cherlin and Point's quantifier elimination procedure

Input : ExpPA formula $\exists \mathbb{x}.\varphi$ with φ quantifier-free

Output: an equivalent quantifier-free formula

$\varphi' \leftarrow$ normalize φ with respect to $\mathbb{x}$ (Sect. 4.1 or [21, First step]), moreover, replace each term $\frac{\mathbb{t}}{d}$ where $d \in \mathbb{Z}^+$ by a fresh variable $\mathbb{x}_j$ and add $d\mathbb{x}_j \leq \mathbb{t} \leq d\mathbb{x}_j + d - 1$ as a conjunct

▷ $\exists \vec{\mathbb{x}} (\in \mathbb{N}^n).\varphi'$ is equivalent with $\exists \mathbb{x}.\varphi$

▷ φ' contains only inequality atomic formulas

$\mathcal{S}_n \leftarrow$ enumerate linear orders among $\mathbb{x}_1, \cdots, \mathbb{x}_n$ (Sect. 4.2 or [21, Second step])

foreach $\sigma \in \mathcal{S}_n$ **do**

  $\varphi'_\sigma \leftarrow \varphi' \wedge \bigwedge_{i \in [n-1]} \mathbb{x}_{\sigma(i)} \leq \mathbb{x}_{\sigma(i+1)}$

  $i \leftarrow n$

  while $i > 0$ **do**

    eliminate exponential occurrences of $\mathbb{x}_{\sigma(i)}$ in φ'_σ (Sect. 4.3.1 or [21, Third step B])

    convert φ'_σ to Disjunctive Normal Form

    eliminate linear occurrences of $\mathbb{x}_{\sigma(i)}$ in φ'_σ ([21, Third step A])

    ▷ variable $\mathbb{x}_{\sigma(i)}$ is removed

    $i \leftarrow i - 1$

  end

  ▷ all quantified variables in φ' are removed

  $\varphi''_\sigma \leftarrow \varphi'_\sigma$

end

return $\bigvee_{\sigma \in \mathcal{S}_n} \varphi''_\sigma$

Algorithm 2 are quite similar. However, the significant difference between them is in the way they handle divisibility constraints.

Cherlin and Point's procedure eliminates all integer division functions during the normalization step by encoding them with fresh variables and inequalities. As a result, when variables are eliminated according to an order $\sigma \in \mathcal{S}_n$, eliminating exponential occurrences of $\mathbb{x}_{\sigma(i)}$ becomes more straightforward because the target formula only contains inequality atomic formulas. However, eliminating linear occurrences of $\mathbb{x}_{\sigma(i)}$ using Cooper's algorithm [23] is not practicable since Cooper's algorithm generates new divisibility constraints containing $\mathbb{x}_{\sigma(i-1)}, \ldots, \mathbb{x}_1$. These new constraints must be encoded with new variables, rendering the algorithm non-terminating.

To address this issue, Cherlin and Point developed a complex sub-procedure to eliminate linear occurrences of $\mathbb{x}_{\sigma(i)}$ from a conjunction of inequalities [22, Theorem 3.3, third step]. This sub-procedure ensures that the new divisibility constraints only contain free variables $\vec{\mathbb{y}}$. The two side effects of using this sub-procedure are: 1) the formula should be converted into Disjunctive Normal Form each time

linear occurrences of $\mathbb{x}_{\sigma(i)}$ are to be eliminated for $i \in [n]$, and 2) even though Cherlin and Point did not conduct a precise complexity analysis of their algorithm, [22] implies that the algorithm's complexity upper-bound is non-elementary at worst. In essence, Cherlin and Point's procedure bounds the original variable $\mathbb{x}$ by a multiple of

$$2^{2^{\cdots^{2^{\mathbb{t}(\mathbb{y})}}}},$$

where $\mathbb{t}(\mathbb{y})$ is a term in φ, and the number of iteration is n, i.e., the number of introduced quantified variables after normalization step [22, p. 6].

We have improved Cherlin and Point's procedure in two ways in this paper. Firstly, inspired by the third step of [22, Thm. 3.3], we designed a sub-procedure to eliminate exponential occurrences of a quantified variable in divisibility atomic formulas locally. Secondly, we incorporated Cooper's algorithm [23] to deal with divisibility atomic formulas in an inductive manner, which also facilitates the complexity analysis in Section 5.

Besides, our presentation is more accessible and natural for computer science researchers because the underlying theory ExpPA is defined more intuitively. In [22], the theory is defined over $\mathbb{N}$, and the subtraction $\dot{-}$ over $\mathbb{N}$ is defined as $\mathbb{x} \dot{-} \mathbb{y} = 0$ if $\mathbb{x} < \mathbb{y}$ and $\mathbb{x} \dot{-} \mathbb{y} = x - y$ is $\mathbb{x} \geq \mathbb{y}$. This definition is rarely used and can be easily confused with subtraction over $\mathbb{Z}$.

5 Complexity analysis

In this section, we analyze the complexity of the proposed quantifier elimination procedure. Consider an existential ExpPA formula of the form

$$\exists \mathbb{x}^1 \cdots \exists \mathbb{x}^m . \varphi(\mathbb{x}^1, \cdots, \mathbb{x}^m)$$

with no free variables. Let n denote the length of the formula. We will prove that the time required to eliminate all its (existential) quantifiers is bounded by $2^{2^{2^{pn \log n}}}$ for some constant p. In other words, the decision problem for existential ExpPA formulas (or equivalently, the satisfiability problem for quantifier-free ExpPA formulas) is in 3-EXPTIME. Up to our knowledge, this is the first upper bound for deciding existential ExpPA formulas.

We first give a brief analysis of the normalization step, in which fresh variables are introduced and the length of the formula grows linearly at most. For the rest of the algorithm, we adopt the strategy of Oppen's analysis of Cooper's algorithm, i.e., to bound the formula length by the product of the number of atomic formulas, the number of coefficients, and the length of the maximum constant (cf. [25]). The critical point of our analysis is that only coefficients of linear occurrences of quantified variables need to be considered.

Normalization

In the normalization step, we normalize the formula with respect to quantified variables from $\mathbb{x}^m$ to $\mathbb{x}^1$. Suppose that fresh variables $\mathbb{x}^i_1, \cdots, \mathbb{x}^i_{N_i}$ are introduced for

each $\mathbb{x}^i (1 \leq i \leq m)$, the formula becomes

$$\exists \mathbb{x}_1^1 \cdots \exists \mathbb{x}_{N_1}^1 \cdots \exists \mathbb{x}_1^m \cdots \exists \mathbb{x}_{N_m}^m . \varphi'(\mathbb{x}_1^1, \cdots, \mathbb{x}_{N_m}^m).$$

Clearly, the number of newly introduced variables is less than the length of the original formula, i.e., we have at most $\sum_{i=1}^{m} N_i \leq n$ quantified variables after normalization.

The increase in the length of the formula during normalization comes from two sources: the additional constraints for introduced variables and the additional formulas for translating $\neq, =$ relations to $\leq$. Both transformations will increase the length of the formula by a constant factor at most.

After normalization, to avoid redundant symbols, we still denote the number of quantified variables by m, and we have $m \leq n$.

Enumeration of linear orders (the outer for-loop in Algorithm 1)

Denote the normalized formula by $\exists \vec{\mathbb{x}}.\varphi(\vec{\mathbb{x}})$ with $\vec{\mathbb{x}} = (\mathbb{x}_1, \ldots, \mathbb{x}_m)$. By Point's procedure, a linear order should be first specified before eliminating quantified variables. Since there are $m!$ possible linear orders for m quantified variables, the outer for-loop terminates in $m!$ iterations at most.

Elimination of quantified variables (the inner for-loop in Algorithm 1)

Suppose the specified linear order is $\mathbb{x}_m \geq \cdots \geq \mathbb{x}_1$. Let $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k}.\varphi_k$ denote the formula obtained by eliminating k quantifiers $\exists \mathbb{x}_m, \ldots, \exists \mathbb{x}_{m-k+1}$ from $\exists \vec{\mathbb{x}}.\varphi(\vec{\mathbb{x}})$. Let $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k}.\varphi'_k$ denote the formula obtained by further eliminating exponential occurrences of $\mathbb{x}_{m-k}$. Let φ_0 denote φ.

Let c_k be the number of distinct divisors in atomic formulas of the form $d \mid \mathbb{t}$ and $d \nmid \mathbb{t}$ plus the number of distinct coefficients of *linear occurrences* of quantified variables in φ_k. Let s_k be the largest constant (including coefficients and divisors) and e_k be the number of atomic formulas in φ_k. Similarly, we define c'_k, s'_k and e'_k for φ'_k.

Example 6 Consider a normalized formula with 2 quantified variables as follows

$$\exists \mathbb{x}_1 \exists \mathbb{x}_2 . 10^{\mathbb{x}_1} + 2\mathbb{x}_2 \leq 200 \wedge 7 | 4 \cdot 10^{\mathbb{x}_2} + 2\mathbb{x}_1 + 3\mathbb{x}_2.$$

Clearly, the formula has two atomic formulas, so $e_0 = 2$. Among all constants in the formula (note that the base 10 is not a constant), 200 is the largest, so $s_0 = 200$. It contains one divisor (i.e., 7) and two distinct coefficients of linear occurrences (i.e., 2 and 3), so $c_0 = 3$.

We analyze the elimination process as follows. Since the formula φ'_k is transformed from φ_k by eliminating exponential occurrences of $\mathbb{x}_{m-k}$, we can bound c'_k, s'_k, e'_k by c_k, s_k, e_k (Lemma 7). Similarly, the formula φ_{k+1} is transformed from φ'_k by eliminating linear occurrences of $\mathbb{x}_{m-k}$, so we can bound $c_{k+1}, s_{k+1}, e_{k+1}$ by c'_k, s'_k, e'_k (Lemma 8). Combining Lemma 7 and Lemma 8, the relation between c_k, s_k, e_k and $c_{k+1}, s_{k+1}, e_{k+1}$ is obtained, and, by induction on k, we can bound c_m, s_m, e_m by c_0, s_0, e_0 (Lemma 9).

Lemma 7 *When the formula* $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k}.\varphi_k$ *is transformed into* $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k}.\varphi'_k$ *by eliminating exponential occurrences of* $\mathbb{x}_{m-k}$*, for* $0 \leq k \leq m-1$*, we have*

$$c'_k \leq {c_k}^2 \qquad s'_k \leq m{s_k}^2 \qquad e'_k \leq 2s_k e_k.$$

Proof Since the elimination of exponential occurrences of variables are local, we only need to discuss two extreme cases where all atomic formulas are of the same form.

When all atomic formulas are inequalities, by Lemma 4, we know that each atomic formula with exponential occurrence of $\mathbb{x}_{m-k}$ is replaced by a new formula. Note that only the coefficients of linear occurrences of $\mathbb{x}_{m-k}$ and $\mathbb{x}_{m-k-1}$ are changed: constant 1 is introduced as a coefficient for $\mathbb{x}_{m-k}$, and if $\mathbb{x}_{m-k}$ is substituted by $\mathbb{x}_{m-k-1} + p$ for some constant p, coefficient of linear occurrence of $\mathbb{x}_{m-k}$ will become $b_{m-k} + b_{m-k-1}$ (b_{m-k}, b_{m-k-1} are coefficients for $\mathbb{x}_{m-k}$ and $\mathbb{x}_{m-k-1}$, see Lemma 4).

- New coefficients of linear occurrences are obtained by adding two linear coefficients together, so we have $c'_k \leq {c_k}^2$.
- Since $\delta \leq \ell_{10}(ms_k)$ and $\gamma \leq \ell_{10}(ms_k)$, when $\mathbb{x}_{m-k}$ is substituted by $\mathbb{x}_{m-k-1} + \delta - 1$ or by $\gamma - 1$, the largest constant in the formula should be less than $s_k 10^{\ell_{10}(ms_k)} \leq m{s_k}^2$, so we have $s'_k \leq m{s_k}^2$.
- From Eq. (7), an inequality atomic formula is replaced by $2\gamma + 2\delta + 1 + 3 + 8 + 1 \leq 4\ell_{10}(ms_k) + 13 \leq 5\ell_{10}(ms_k)$ atomic formulas.

When all atomic formulas are divisibility atomic formulas of the form $d|\mathbb{t}$ or $d \nmid \mathbb{t}$, by Section 4.3.2, a divisibility atomic formula $d|\mathbb{t}$ with exponential occurrences of $\mathbb{x}_{m-k}$ is replaced by a new formula.

- Coefficients of linear occurrences of variables remain unchanged, and a divisibility atomic formula produces at most two forms of divisibility atomic formulas $d|\mathbb{t}'$ and $\phi(d)|\mathbb{t}'$. So we have $c'_k \leq c_k$.
- In a divisibility atomic formula, any constant in the divident $\mathbb{t}$, say l, can be replaced by $(l \bmod d)$. So we have $s'_k \leq s_k$.
- When d is a prime number with $d \neq 2$ or 5, we have $d = d_0$ and $\phi(d_0) = d-1$. In this case, from Eq. (9), we know that a divisibility atomic formula is replaced by $1+1+2\phi(d_0) = 2d$ atomic formulas, so we have $e'_k \leq 2s_k e_k$.

Choose larger upper bounds for c'_k, s'_k and e'_k respectively, then the lemma is proved. □

When φ'_k is transformed into φ_{k+1} by eliminating linear occurrences of $\mathbb{x}_m$, Oppen's analysis gives the following lemma:

Lemma 8 ([25, p.35]) *When the formula* $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k}.\varphi'_k$ *is transformed into* $\exists \mathbb{x}_1 \ldots \exists \mathbb{x}_{m-k-1}.\varphi_{k+1}$ *by eliminating linear occurrences of* $\mathbb{x}_{m-k}$*, for* $0 \leq k \leq m-1$*, we have*

$$c_{k+1} \leq {c'_k}^4 \qquad s_{k+1} \leq {s'_k}^{4c'_k} \qquad e_{k+1} \leq {e'_k}^4 {s'_k}^{2c'_k}.$$

Combining Lemma 7 and 8, we have, for $0 \leq k \leq m-1$,

$$c_{k+1} \leq {c_k}^8 \qquad s_{k+1} \leq (m{s_k}^2)^{4{c_k}^2} \qquad e_{k+1} \leq (2s_k e_k)^4 (m{s_k}^2)^{2{c_k}^2}.$$

Therefore, the following bounds are obtained.

Lemma 9 *For* $0 \leq k \leq m-1$,

$$c_k \leq {c_0}^{8^k} \qquad s_k \leq m^{(4c_0)^{8^k}} {s_0}^{(8c_0)^{8^k}} \qquad e_k \leq 2^{4^{2k}} {e_0}^{4^k} m^{(4c_0)^{8^k}} s_0^{(8c_0)^{8^k}}. \tag{10}$$

Proof We prove by induction on k:

Basic case. It is easy to verify that Eq. (10) holds when $k = 0$.

Inductive step. Let us suppose Eq. (10) holds for some $k \in \mathbb{N}$, we prove that it still holds for $k+1$.

$$c_{k+1} \leq {c_k}^8 \leq ({c_0}^{8^k})^8 \leq {c_0}^{8^{k+1}},$$

$$\begin{aligned}
s_{k+1} &\leq (m{s_k}^2)^{4{c_k}^2} \\
&\leq m^{4{c_k}^2} \cdot {s_k}^{8{c_k}^2} \\
&\leq m^{4{c_0}^{2\cdot 8^k}} (m^{(4c_0)^{8^k}} {s_0}^{(8c_0)^{8^k}})^{8{c_0}^{2\cdot 8^k}} \\
&\leq m^{4{c_0}^{2\cdot 8^k} + (4c_0)^{8^k} \cdot 8{c_0}^{2\cdot 8^k}} {s_0}^{(8c_0)^{8^k} \cdot 8{c_0}^{2\cdot 8^k}} \\
&\leq m^{((4c_0)^{8^k})^8} {s_0}^{((8c_0)^{8^k})^8} \\
&= m^{(4c_0)^{8^{k+1}}} {s_0}^{(8c_0)^{8^{k+1}}},
\end{aligned}$$

$$\begin{aligned}
e_{k+1} &\leq (2s_k e_k)^4 (m{s_k}^2)^{2{c_k}^2} \\
&\leq 2^4 {e_k}^4 m^{2{c_k}^2} {s_k}^{4+4{c_k}^2} \\
&\leq 2^4 \left(2^{4^{2k}} {e_0}^{4^k} m^{(4c_0)^{8^k}} s_0^{(8c_0)^{8^k}}\right)^4 m^{2({c_0}^{8^k})^2} \left(m^{(4c_0)^{8^k}} {s_0}^{(8c_0)^{8^k}}\right)^{4+4({c_0}^{8^k})^2} \\
&\leq 2^{4^{2k+2}} {e_0}^{4^{k+1}} m^{((4c_0)^{8^k})^8} {s_0}^{((8c_0)^{8^k})^8} \\
&= 2^{4^{2(k+1)}} {e_0}^{4^{k+1}} m^{(4c_0)^{8^{k+1}}} s_0^{(8c_0)^{8^{k+1}}}.
\end{aligned}$$

□

Recall that n denotes the length of the formula, we make following assumptions: $c_0 \leq n$, $s_0 \leq n$, $e_0 \leq n$, and $m \leq n$. The space required to store the resulting quantifier free formula is bounded by the product of the number of linear orders $m!$, the number of atomic formulas e_m, the maximum number of constants $2m+2$ per atom, the maximum amount of space $\log s_m$ to store each constant and some constant p,

that is,

$$\begin{aligned}\mathsf{space} =& m! \cdot e_m \cdot (2m+2) \cdot \log s_m \cdot p \\ =& {e_m}^{O(1)} \\ =& (2^{4^{2m}} {e_0}^{4^m} m^{(4c_0)^{8^m}} s_0^{(8c_0)^{8^m}})^{O(1)} \\ =& (2^{2^{4n}} n^{4^n} n^{(4n)^{8^n}} (n)^{(8n)^{8^n}})^{O(1)} \\ =& ((n)^{(8n)^{8^n}})^{O(1)} \\ =& 2^{O((8n)^{8^n} \log n)} \\ =& 2^{2^{O(n \log n)}}\end{aligned} \tag{11}$$

where $O(\cdot)$ is the "big-O" asymptotic notation, i.e., $f(n) = O(g(n))$ is defined as $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$ for some constant c. Since the upper bound on deterministic time is dominated by the square of the time required to write out φ_m [25], the space bound is also a bound on deterministic time. We summarize our analysis as follows:

Theorem 10 *The decision procedure for existential* ExpPA*, using* ExpPA *quantifier elimination procedure Algorithm 1, has a triply exponential upper bound on both deterministic time and space (3-EXPTIME and 3-EXPSPACE).*

Remark 2 (Unary encoding vs. binary encoding) In our definition of ExpPA, linear terms are expressed using the unary encoding. For instance, $4\mathbb{x}$ are encode by $\mathbb{x} + \mathbb{x} + \mathbb{x} + \mathbb{x}$. In fact, we can extend ExpPA to admit terms of the form $k \cdot \mathbb{x}$ where k is encoded with a binary representation. If so, we should change the assumption $s_0 \leq n$ to $s_0 \leq 2^n$, one can check that this will not affect the result in Eq. (11). (Alternatively, one can simulate binary encoding by introducing additional polynomial many variables [26, p. 3].)

Remark 3 (Tightness) The goal of this section is to show the decision problem for existential ExpPA has an elementary complexity upper bound. It is noteworthy that our algorithm employs Cooper's algorithm as a subprocedure; however, the complexity of Cooper's algorithm is also 3-EXPTIME [25]. This observation indicates that the bottleneck in efficiency lies in finding more efficient approaches to handle linear formulas. We conjecture that employing a more efficient algorithm (or conducting a more fine-grained analysis) for processing linear formulas could yield tighter complexity results. Moreover, the known complexity lower bound is NP, as the satisfiability problem for existential PA formulas is NP-complete.

Remark 4 (Alternative quantifiers) Note that our analysis is focused on existential ExpPA formulas and is sufficient for our string constraints solving context. Unfortunately, directly extending our analysis on general ExpPA formulas with alternating quantifiers, by transforming $\forall \mathbb{x}.\varphi$ into $\neg\exists\mathbb{x}.\neg\varphi$, will not give an elementary upper bound.

6 Optimizations

In the last section we showed the complexity of Algorithm 1 over existential ExpPA formulas to be triply exponential, which is quite expensive and a faithful implementation would not scale [1]. Note that this high complexity holds even for quantifier-free ExpPA formulas: for a quantifier-free formula φ, we solve its satisfiability problem by adding the existential quantifiers for all the variables occurring in φ, then eliminate the quantifiers one by one, resulting in true or false in the end. The original formula φ is satisfiable if true is obtained in the end.

In this section, we focus on quantifier-free ExpPA formulas (or existential ExpPA formulas since they are satisfiability-equivalent), and present various optimizations of the quantifier elimination procedure for ExpPA, aiming at an efficient implementation. The focus on quantifier-free ExpPA formulas is motivated by the following two facts: 1) the flattening of $\mathsf{STR}_{\mathsf{parseInt}}$ constraints results into such formulas, 2) these formulas are already challenging for state-of-the-art SMT solvers (with exponential functions defined as recursive functions).

Let φ be a quantifier-free ExpPA formula in the remainder of this section. Moreover, we assume that φ is normalized since the optimizations presented in the sequel are for normalized formulas. Furthermore, for technical convenience, we assume that all the inequality atomic formulas are of the form $\sum_{j=1}^{n} a_j 10^{\mathbb{x}_j} + \sum_{k=1}^{n} b_k \mathbb{x}_k \leq c$, where c is an integer constant. (Implicitly, we assume that there are no free variables and all the variables are existentially quantified.)

In the sequel, we will explain two major optimizations in Section 6.1 and Section 6.2. Additional optimizations are listed in Section 6.3.

6.1 Reducing the number of enumerated variable orders by over-approximation

Recall that in Point's procedure for ExpPA, after the normalization, the variable orders are enumerated and for each order, the exponential and linear occurrences of variables are eliminated. Since the quantifier elimination is expensive and applied to each possible order of variables, if we could reduce the candidate variable orders in the very beginning, it would facilitate considerable speed-up for the decision procedure.

The main idea is to consider an over-approximation of φ, which is a PA formula φ', and use φ' to remove the infeasible candidate variable orders. Note that all the exponential terms in φ is of the form $10^{\mathbb{z}}$ for some integer variable $\mathbb{z}$. The over-approximation is based on the observation that $10^n \geq 9n + 1$ for every $n \in \mathbb{N}$. We obtain the over approximation φ' from φ by replacing each exponential term $10^{\mathbb{z}}$ with a fresh variable $\mathbb{z}'$ and add $\mathbb{z}' \geq 9\mathbb{z} + 1$ as a conjunct. Then during the enumeration of the linear orders for the variables $\mathbb{x}_1, \ldots, \mathbb{x}_n$, we can quickly remove those infeasible candidates σ such that $\varphi' \wedge \bigwedge_{i \in [n-1]} \mathbb{x}_{\sigma(i)} \leq \mathbb{x}_{\sigma(i+1)}$ is unsatisfiable. A special case is that if φ' itself is unsatisfiable, then we can directly conclude that the original formula φ is unsatisfiable. In our implementation, the unsatisfiability is checked

[1] We did implement Algorithm 1 and discovered that the implementation could only solve formulas of very small size.

over $\mathbb{R}$ instead of $\mathbb{N}$, since checking the satisfiability of an existential PA formula is still a NP-complete problem.

6.2 Avoiding the elimination of linear occurrences of variables by under approximation

The decision procedure of ExpPA in Algorithm 1 requires the elimination of both exponential and linear occurrences of variables. Considering the fact that PA formulas can be solved efficiently by the state-of-the-art solvers, e.g., CVC4 and Z3, one natural idea is to try to only eliminate the exponential occurrences, but not the linear occurrences, of variables, and obtain the PA formulas in the end, which can then be solved by the state-of-the-art solvers.

Recall that Lemma 4 enables us to eliminate the exponential occurrences of $\mathbb{x}_{\sigma(i)}$ from an atomic formula τ of the form

$$\tau(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}) \hat{=} \\ a_i 10^{\mathbb{x}_{\sigma(i)}} + \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i} b_k \mathbb{x}_{\sigma(k)} \leq c.$$

Actually, Lemma 4 does more in the sense that all occurrences of $\mathbb{x}_{\sigma(i)}$, including the linear ones, are eliminated from the atomic formulas resulted from τ, e.g., $\tau[\mathbb{x}_{\sigma(i-1)} + p/\mathbb{x}_{\sigma(i)}]$. Then we can continue eliminating the exponential occurrences of $\mathbb{x}_{\sigma(i-1)}$ from $\tau[\mathbb{x}_{\sigma(i-1)} + p/\mathbb{x}_{\sigma(i)}]$, provided that the coefficient of $\mathbb{x}_{\sigma(i-1)}$ therein is nonzero. Iterating this process would produce a PA formula eventually.

Nevertheless, the condition of Lemma 4, namely $a_i \neq 0$, hinders the aforementioned natural idea. If $a_i = 0$, but $b_i \neq 0$, then we are unable to utilize Lemma 4 to eliminate the linear occurrences of $\mathbb{x}_{\sigma(i)}$ from τ. In this case, the quantifier elimination algorithm of PA has to be applied to eliminate $\mathbb{x}_{\sigma(i)}$, so that later on, we can eliminate the exponential occurrences of $\mathbb{x}_{\sigma(i-1)}$, which requires that $\mathbb{x}_{\sigma(i-1)}$ is the maximum variable in the left-hand side of the inequality.

To avoid applying the quantifier elimination algorithm of PA, we consider the following under approximation of τ, namely, we additionally assume that $\mathbb{x}_{\sigma(i)} \leq 10^u$ for some constant bound $u \in \mathbb{N}$. When $a_i = 0$, τ can be rewritten as

$$\tau'(\mathbb{x}_{\sigma(i)}, \ldots, \mathbb{x}_{\sigma(1)}) \hat{=} \sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i-1} b_k \mathbb{x}_{\sigma(k)} \leq c - b_i \mathbb{x}_{\sigma(i)}.$$

Let us assume $a_{i-1} > 0$. Define c_1, c_2 as follows: If $b_i > 0$, then $c_1 = c - b_i 10^u$ and $c_2 = c$, otherwise, $c_1 = c$ and $c_2 = c - b_i 10^u$. It is easy to observe that $c_1 \leq c - b_i \mathbb{x}_{\sigma(i)} \leq c_2$. Then we can apply Lemma 4 to the following two inequalities to eliminate $10^{\mathbb{x}_{\sigma(i-1)}}$,

$$\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i-1} b_k \mathbb{x}_{\sigma(k)} \leq c_1 \tag{12}$$

and

$$\sum_{j=1}^{i-1} a_j 10^{\mathbb{x}_{\sigma(j)}} + \sum_{k=1}^{i-1} b_k \mathbb{x}_{\sigma(k)} \leq c_2. \tag{13}$$

Let $\alpha_1 = \ell_{10}(c_1) - \ell_{10}(a_{i-1})$ and $\alpha_2 = \ell_{10}(c_2) - \ell_{10}(a_{i-1})$. Then from Lemma 4, let $A \hat{=} 1 + \sum_{j=1}^{i-2} |a_j|$, $B \hat{=} 1 + \sum_{j=1}^{i-1} |b_j|$, $\delta \hat{=} \ell_{10}(A) + 2$, and $\gamma \hat{=} 2\ell_{10}(B) + 3$. For a given interpretation I such that $I \models \mathbb{x}_{\sigma(i-1)} \geq \gamma \wedge \mathbb{x}_{\sigma(i-1)} \geq \mathbb{x}_{\sigma(i-2)} + \delta$

- if α_1 or α_2 is undefined, i.e., $I \models \alpha_1 \leq 0 \vee \alpha_2 \leq 0$, then $I \not\models \tau'$;
- if $I \models \mathbb{x}_{\sigma(i-1)} \geq \alpha_2 + 2$, then $I \not\models$ Eq. (13), thus $I \not\models \tau'$.
- if $I \models \mathbb{x}_{\sigma(i-1)} \leq \alpha_1 - 1$, then $I \models$ Eq. (12), thus $I \models \tau'$.

Therefore, τ' and τ are equivalent to

$$\begin{aligned}
&\bigvee_{p=0}^{\gamma-1} (\mathbb{x}_{\sigma(i-1)} = p \wedge \tau[p/\mathbb{x}_{\sigma(i-1)}]) \\
\vee &\bigvee_{p=0}^{\delta-1} \left(\mathbb{x}_{\sigma(i-1)} = \mathbb{x}_{\sigma(i-2)} + p \wedge \tau[\mathbb{x}_{\sigma(i-2)} + p/\mathbb{x}_{\sigma(i-1)}] \right) \\
\vee &\left(\mathbb{x}_{\sigma(i-1)} \geq \gamma \wedge \mathbb{x}_{\sigma(i-1)} \geq \mathbb{x}_{\sigma(i-2)} + \delta \wedge \mathbb{x}_{\sigma(i-1)} \leq \alpha_1(\vec{\mathbb{y}}) - 1 \right) \\
\vee &\bigvee_{p=\alpha_1}^{\alpha_2+1} \begin{pmatrix} \mathbb{x}_{\sigma(i-1)} \geq \gamma \wedge \mathbb{x}_{\sigma(i-1)} \geq \mathbb{x}_{\sigma(i-2)} + \delta \\ \wedge \mathbb{x}_{\sigma(i-1)} = p \wedge \tau[\alpha(\vec{\mathbb{y}}) + p/\mathbb{x}_{\sigma(i-1)}] \end{pmatrix}
\end{aligned} \tag{14}$$

where all exponential occurrences of $\mathbb{x}_{\sigma(i-1)}$ are eliminated. Note that in the last line of Eq. (14), $\mathbb{x}_{\sigma(i-1)}$ is enumerated from α_1 to $\alpha_2 + 1$, while in Eq. (7) only two term $\alpha(\vec{\mathbb{y}})$ and $\alpha(\vec{\mathbb{y}}) + 1$ are enumerated for $\mathbb{x}_{\sigma(i)}$.

Similarly, we can eliminate the exponential occurrences of $\mathbb{x}_{\sigma(i-2)}$ from formula $\tau'[\mathbb{x}_{\sigma(i-2)} + p/\mathbb{x}_{\sigma(i-1)}]$ as well as $\tau'[p/\mathbb{x}_{\sigma(i-1)}]$, and so on. Eventually, we obtain a PA formula.

6.3 Additional optimization techniques

Eliminating exponential occurrences in atomic formulas simultaneously

Although Lemma 4 is stated for a single atomic formula, the elimination of the exponential occurrences of the same variable in different atomic formulas can actually be conducted simultaneously. That is, let $\alpha_1^\tau, \alpha_2^\tau, \gamma^\tau, \delta^\tau$ be the constants as stated in the aforementioned under-approximation of an inequality τ, define $\alpha_1^{\min}, \alpha_2^{\max}, B^{\max}, \delta^{\max}$ as the minimum of α_1^τ, the maximum of α_2^τ, the maximum of B^τ, and the maximum of δ^τ respectively with τ ranging over the inequalities of φ. Then we can use the same constants $\alpha_1^{\min}, \alpha_2^{\max}, B^{\max}, \delta^{\max}$ for different inequalities when eliminating the exponential occurrences of the same variable.

Avoiding the formula-size blow-up by depth-first search

The PA formula resulting from the elimination of exponential occurrences is essentially a disjunction of large number of disjuncts of small size. If we store this large formula naively, then its size quickly blows up and exhausts the memory. Alternatively, we choose to do a depth-first search (DFS) and consider the disjuncts, which are of small sizes, one by one, and solve the satisfiability problem for these disjuncts. When a satisfiable disjunct is found, then the search terminates and "SAT" is reported.

Preprocessing with small upper bound

We believe that if a quantifier-free ExpPA formula is satisfiable, then more likely it may be satisfied with an assignment in which all variables are uniformly bounded. Consequently, as a preprocessing step, we put a small upper bound, e.g. the biggest constant occurring in the formula, on the values of variables, and perform a depth-first search, so that a model can be quickly found if there is any. If this preprocessing is unsuccessful, then we continue the search with the greater upper bound 10^u for some proper predefined $u \in \mathbb{N}$.

7 Implementation and experiments

7.1 Implementation

We implement the decision procedure in Wolfram Mathematica, called EXP-solver, which is able to solve the satisfiability of ExpPA formulas.

EXP-solver takes a quantifier-free ExpPA formula as input. Moreover, it allows specifying a upper bound 10^u to uniformly bound the values of variables: given a upper bound 10^u, the problem is to decide whether there is an assignment in which all variable values are bounded by 10^u satisfying the given formula. Outputs of EXP-solver are either "SAT", "UNSAT", "B-UNSAT", or "TIMEOUT", standing for the given formula satisfiable, unsatisfiable, unsatisfiable up to 10^u, or timeout if the solver does not terminate within the time limit. If the output is "SAT", then a model (namely, an assignment) is returned.

7.2 Benchmarks

To evaluate the performance of EXP-solver, we design two benchmark suites, ARITHMETIC and STRINGHASH[2].

The ARITHMETIC benchmark suite

This suite comprises three groups of randomly generated ExpPA formulas. Each group is characterized by four parameters (EV, LV, EI, LI), where EV, LV represent the number of variables with exponential occurrences and with only linear occurrences respectively, and EI, LI represent the number of inequalities with exponential terms and with only linear terms respectively. We consider three parameter

[2]The benchmarks are available at https://github.com/EcstasyH/EXP-solver

classes, $(2,3,3,4)$, $(3,4,4,5)$, and $(4,5,5,6)$. Each group of the benchmark suite consists of 200 randomly generated problem instances. The coefficients of exponential terms are randomly selected from the interval $[-10^2, 10^2]$ and the other coefficients/constants are randomly selected from $[-10^5, 10^5]$. The two intervals are chosen with the intention that the coefficients of exponential terms are smaller so that they do not always dominate the left-hand side of the inequalities. Moreover, aiming at a better coverage of the syntactical ingredients of ExpPA, we randomly choose some problem instances and replace the $\leq$ symbol of their first inequalities by $=$. The constant upper bound for the values of variables is set to be 10^{20}, as the largest 64-bit integer is less than 10^{20}. We also create an SMTLib2 file for each problem instance, to facilitate the comparison with the state-of-the-art of SMT solvers CVC4 and Z3. Because neither CVC4 nor Z3 supports the exponential functions directly, in the SMTLib2 files, we encode 10^x as a recursive function $f(x)$ defined by: $f(0) = 1$ and $f(n+1) = 10 \cdot f(n)$.

The STRINGHASH benchmark suite

This suite comprises two groups of string constraints generated from the string hash functions $\mathsf{hash} : \Sigma^*_{\mathsf{num}} \to \mathbb{Z}$ encoded by $\mathsf{parseInt}$. Specifically, the hash function is defined as

$$\mathsf{hash}(x) = (\mathsf{parseInt}(x) \bmod m) \bmod m',$$

where $m, m' \in \mathbb{N}$ are user-defined. The string constraints in the STRINGHASH benchmark suite are of the form

$$x \in \mathcal{A} \wedge (\mathsf{parseInt}(x) \bmod m) \bmod m' = 0 \wedge \mathsf{len}(x) < 100,$$

where $\mathcal{A}$ is an FA, m is a randomly chosen prime number in the interval $[10^2, 10^5]$ and m' is a number less than m (not necessarily prime). Intuitively, we try to find a string x following pattern $\mathcal{A}$ with hash value 0 and length less than 100.

We restrict x in one group conforming some flat patterns (the flat group), while for the other, we allow more general patterns (the non-flat group). The flat group comprises 300 problem instances, where the flat languages are of the form $12345w_1^+w_2^+$, $12345w_1^+w_2^+6789$, or $w_1^+w_2^+6789$, with $w_1, w_2 \in \Sigma^+_{\mathsf{num}}$. The non-flat group comprises 300 problem instances, where the non-flat languages are of the form $12345\Sigma^*_{\mathsf{num}}$, $12345\Sigma^*_{\mathsf{num}}6789$, or $\Sigma^*_{\mathsf{num}}6789$. The constraints can be thought of as 12345 and 6789 are the text to be protected, and $w_1^+w_2^+$ or Σ^*_{num} is the pattern for some nonce string.

We generate the SMTLib2 files for these string constraints, as inputs to the string constraint solvers. On the other hand, for EXP-solver, we do the following:

- For flat instances, we generate ExpPA formulas corresponding to the string constraints, as inputs to EXP-solver.
- For non-flat instances, we use flat languages $a^*(b_1 \dots b_k)$ to under-approximate Σ^*_{num}, where $a, b_1, \dots, b_k \in \Sigma_{\mathsf{num}}$. We iterate the following procedure until a model is found or timeout: Initially, set $k = 1$ and iterate by assigning $0, \dots, 9$ to a. For each assignment, we encode the resulting string constraint into an ExpPA

formula with only one exponential variable. If the resulting ExpPA formula is unsatisfiable, then we increase k by 1 and repeat this process.

We would like to remark that the flattening strategy for non-flat regular constraints here is a strict generalization of that in [17]: Patterns of the form $0^*(b_1...b_k)$ were considered therein and PA formulas are sufficient to encode such patterns. On the other hand, we consider patterns of the form e.g. $(a)^*(b_1 \dots b_k)$ (where $a \in \Sigma_{\mathsf{num}}$ can be nonzero), which requires ExpPA formulas to encode in general.

7.3 Experiments

We compare EXP-solver with the state-of-the-art SMT solvers on the generated benchmarks. Specifically,

- for the ARITHMETIC benchmark suite, we compare EXP-solver against Z3 (version 4.8.10), CVC4 (version 1.8), and CVC5 (version 1.0.1)[3],
- for the STRINGHASH benchmark suite, we compare EXP-solver against Z3, CVC4, CVC5, and Trau[4].

All the experiments are run on a lap-top with the Intel i5 1.4GHz CPU and 8GB memory. We set the time limit as 60 seconds per problem instance.

The experiment results are summarized in Table 1. For Z3 and CVC4, "Fail" means timeout or unknown; for Trau, "Fail" means timeout, unknown or wrong answers [5]; for EXP-solver, "Fail" only means timeout.

For the ARITHMETIC benchmark suite, EXP-solver solves around 20%-60% more instances than Z3, and 30%-100% more instances than CVC4. Moreover, the gap becomes bigger as the the sizes of the formulas increase, which demonstrates that EXP-solver is more efficient in solving formulas of greater sizes. The average time of EXP-solver is comparable with Z3 and CVC4. EXP-solver reports "B-UNSAT" for 47 instances of the $(2, 3, 3, 4)$-group, while it does not report "B-UNSAT" (except one) for the other two groups. If more time is allowed, EXP-solver is able to report "B-UNSAT" for the "TIMEOUT" instances.

For the STRINGHASH benchmark suite, in overall, EXP-solver solves significantly more instances, especially those satisfiable instances, than Z3, CVC4, and Trau. For instance, for flat regular constraints, EXP-solver solves almost all 300 problem instances, except 3 of them[6], while Z3, CVC4, Trau solve only 34, 89, 187 instances respectively. Trau gets wrong answers for some problem instances, e.g. it reports "UNSAT" for some satisfiable instances. From the results, we can see that EXP-solver achieves a good tradeoff between precision and efficiency, although it is slower than the other solvers.

[3] We omit CVC5 results in the following because they are similar to CVC4 results. One can find more details through the link we have given above.

[4] https://github.com/guluchen/z3/tree/new_trau

[5] In satisfiable instances, the assignments given by the tools can be verified using large number calculations (supported by Wolfram Mathematica). We found that, due to some unknown reasons, Trau may return wrong answers in the stringhash benchmark suite.

[6] These three instances can actually be solved in 70 seconds.

Table 1 Experimental Results, left: results for both experiments, right: more detailded results of STRINGHASH benchmark suite. O: Output, S:SAT, U: UNSAT, B: Bounded UNSAT, F: Fail, #: number of problems, T: average time in seconds, TO: timeout (60s), "-": unsupported or meaningless.

Group	O	Z3		CVC4		Trau		EXP-solver	
		#	T	#	T	#	T	#	T
(2,3,3,4)	S	56	**0.4**	42	2.3	-	-	**64**	**0.4**
	U	69	**0.1**	72	**0.1**	-	-	**89**	**0.1**
	B	-	-	-	-	-	-	47	9.5
	F	75	TO	86	TO	-	-	**0**	TO
(3,4,4,5)	S	33	**1.4**	25	2.9	-	-	**52**	3.3
	U	59	**0.1**	60	**0.1**	-	-	**88**	**0.1**
	B	-	-	-	-	-	-	1	54.0
	F	108	TO	115	TO	-	-	**59**	TO
(4,5,5,6)	S	35	**1.8**	19	6.6	-	-	**47**	22.4
	U	36	0.3	39	0.4	-	-	**72**	**0.1**
	B	-	-	-	-	-	-	0	-
	F	129	TO	142	TO	-	-	**81**	TO
Flat	S	34	19.0	88	12.7	5	**0.1**	**115**	12.3
	U	0	-	1	4.0	182	**2.5**	**182**	47.7
	F	266	TO	211	TO	113	TO	**3**	TO
Non-flat	S	210	7.8	144	4.9	55	5.9	**300**	16.7
	U	0	-	0	-	0	-	0	-
	F	90	TO	156	TO	245	TO	**0**	TO

Group	O	Z3		CVC4		Trau		EXP-solver	
		#	T	#	T	#	T	#	T
$12345(w_1)^+(w_2)^+$	S	5	14.0	29	8.5	3	**0.1**	**37**	9.9
	U	0	-	0	-	**60**	**1.3**	**60**	47.2
	F	95	TO	71	TO	37	TO	**3**	TO
$12345(w_1)^+(w_2)^+6789$	S	11	13.0	29	12.0	0	-	**37**	**10.6**
	U	0	-	0	-	**63**	**1.2**	**63**	50.0
	F	89	TO	71	TO	37	TO	**0**	TO
$(w_1)^+(w_2)^+6789$	S	18	24.0	30	9.3	2	**0.1**	**41**	16.1
	U	0	-	1	4.0	**59**	**2.5**	**59**	45.8
	F	82	TO	69	TO	39	TO	**0**	TO
$12345\Sigma^*_{\text{num}}$	S	82	8.7	**100**	**2.2**	28	5.9	**100**	18.5
	U	0	-	0	-	0	-	0	-
	F	18	TO	**0**	TO	72	TO	**0**	TO
$12345\Sigma^*_{\text{num}}6789$	S	60	9.3	17	7.8	3	**0.3**	**100**	16.0
	U	0	-	0	-	0	-	0	-
	F	40	TO	83	TO	97	TO	**0**	TO
$\Sigma^*_{\text{num}}6789$	S	68	5.5	27	13.0	24	**9.0**	**100**	15.7
	U	0	-	0	-	0	-	0	-
	F	32	TO	73	TO	76	TO	**0**	TO

8 Conclusion

In this paper, we proposed a complete flattening approach for string constraints with string-integer conversion and flat regular constraints, based on a new quantifier elimination procedure for ExpPA, i.e., the extension of Presburger arithmetic with exponential functions. The new procedure is improved from the original quantifier elimination procedure by Cherlin and Point in 1986. We also analyze its complexity and for the first time show a 3-EXPTIME complexity upper bound for the decision problem of existential ExpPA formulas. Moreover, we proposed various optimizations and achieved the first prototypical implementation. We also did extensive experiments to evaluate the performance of the implementation. The experiment results show the efficacy of our implementation, compared with the state-of-the-art solvers.

References

[1] Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)

[2] Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science, FOCS'99, pp. 495–500. IEEE Computer Society, Washington, DC (1999). https://doi.org/10.1109/SFFCS.1999.814622

[3] Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification - 23rd International Conference, CAV'11. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14

[4] Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS'08. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

[5] Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, pp. 114–124. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491456

[6] Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1232–1243. ACM, New York, NY, US (2014). https://doi.org/10.1145/2660267.2660372

[7] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Computer Aided Verification - 27th International Conference, CAV'15. Lecture Notes in Computer Science, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29

[8] Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proceedings of the ACM on Programming Languages **2**(POPL), 3–1329 (2018) https://doi.org/10.1145/3158091

[9] Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proceedings of the ACM on Programming Languages **2**(POPL), 4–1432 (2018) https://doi.org/10.1145/3158092

[10] Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T., Yu,

F.: Parameterized model counting for string and numeric constraints. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE'18, pp. 400–410. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236064

[11] Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS'10. Lecture Notes in Computer Science, vol. 6015, pp. 154–157. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13

[12] Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Bjørner, N.S., Gurfinkel, A. (eds.) Formal Methods in Computer Aided Design, FMCAD'18, pp. 1–5. IEEE, Washington, DC (2018). https://doi.org/10.23919/FMCAD.2018.8602997

[13] Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of word equations: Decidable and undecidable theories. In: Reachability Problems - 12th International Conference, RP'18. Lecture Notes in Computer Science, vol. 11123, pp. 15–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00250-3_2

[14] ECMAScript, E., Association, E.C.M., et al.: ECMAScript language specification (2019). https://www.ecma-international.org/ecma-262/

[15] Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology, **21**(4), 25–12528 (2012) https://doi.org/10.1145/2377656.2377662

[16] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 31st IEEE Symposium on Security and Privacy, S&P'10, pp. 513–528. IEEE Computer Society, Washington, DC (2010). https://doi.org/10.1109/SP.2010.38

[17] Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janku, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI'20, pp. 943–957. ACM, New York, NY, US (2020). https://doi.org/10.1145/3385412.3386034

[18] Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'17, pp. 602–617. ACM, New York,

NY, US (2017). https://doi.org/10.1145/3062341.3062384

[19] Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC'13. Lecture Notes in Computer Science, vol. 8244, pp. 15–31. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_2

[20] Semënov, A.L.: Logical theories of one-place functions on the set of natural numbers. Mathematics of the USSR-Izvestiya **22**(3), 587–618 (1984) https://doi.org/10.1070/IM1984v022n03ABEH001456

[21] Cherlin, G., Point, F.: On extensions of presburger arithmetic. In: Proceedings of the Fourth Easter Conference on Model Theory, Gross Koris, pp. 17–34 (1986). https://webusers.imj-prg.fr/%7efrancoise.point/papiers/cherlin_point86.pdf

[22] Point, F.: On the expansion $(\mathbb{N}, +, 2^x)$ of Presburger arithmetic. (2007). Unpublished. https://webusers.imj-prg.fr/%7francoise.point/papiers/Pres.pdf

[23] Cooper, D.C.: Theorem proving in arithmetic without multiplication. Machine intelligence **7**, 91–100 (1972)

[24] Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers (6. Ed.). Clarendon Press, Oxford (2008)

[25] Oppen, D.C.: Elementary bounds for presburger arithmetic. In: Proceedings of the 5th Annual ACM Symposium on Theory of Computing, STOC'73, pp. 34–37. ACM, New York, NY, US (1973). https://doi.org/10.1145/800125.804033

[26] Haase, C.: Subclasses of presburger arithmetic and the weak EXP hierarchy. In: Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014, pp. 47–14710. ACM, New York, NY, US (2014). https://doi.org/10.1145/2603088.2603092