Counting Algorithms for Recognizable and Algebraic Series

Bao Trung CHU†[a], Nonmember, Kenji HASHIMOTO†, and Hiroyuki SEKI†, Members

SUMMARY  Formal series are a natural extension of formal languages by associating each word with a value called a coefficient or a weight. Among them, recognizable series and algebraic series can be regarded as extensions of regular languages and context-free languages, respectively. The coefficient of a word $w$ can represent quantities such as the cost taken by an operation on $w$, the probability that $w$ is emitted. One of the possible applications of formal series is the string counting in quantitative analysis of software. In this paper, we define the counting problems for formal series and propose algorithms for the problems. The membership problem for an automaton or a grammar corresponds to the problem of computing the coefficient of a given word in a given series. Accordingly, we define the counting problem for formal series in the following two ways. For a formal series $S$ and a natural number $d$, we define $CC(S, d)$ to be the sum of the coefficients of all the words of length $d$ in $S$ and $SC(S, d)$ to be the number of words of length $d$ that have non-zero coefficients in $S$. We show that for a given recognizable series $S$ and a natural number $d$, $CC(S, d)$ can be computed in $O(\eta \log d)$ time where $\eta$ is an upper-bound of time needed for a single state-transition matrix operation, and if the state-transition matrices of $S$ are commutative for multiplication, $SC(S, d)$ can be computed in polynomial time of $d$. We extend the notions to tree series and discuss how to compute them efficiently. Also, we propose an algorithm that computes $CC(S, d)$ in square time of $d$ for an algebraic series $S$. We show the CPU time of the proposed algorithm for computing $CC(S, d)$ for some context-free grammars as $S$, one of which represents the syntax of C language. To examine the applicability of the proposed algorithms to string counting for the vulnerability analysis, we also present results on string counting for Kaluza Benchmark.

key words: string counting, recognizable series, algebraic series, context-free grammar

1. Introduction

Formal power series (or formal series for short) are a natural extension of formal languages by assigning to each word (or string) in a language a value called the coefficient (or the weight) of the word taken from a semiring. A simple and important subclass of formal series is recognizable series, which can be regarded as an extension of regular languages. A recognizable series $S$ is represented by a triple $(\lambda, \mu, \gamma)$ where $\lambda$ is a row vector specifying the initial states, $\mu$ is a morphism from input words to state-transition matrices, and $\gamma$ is a column vector specifying the final states. The coefficient of a word $w$ is defined as $(S, w) = \lambda(\mu w) \gamma$. The support of a formal series is the language consisting of words $w$ such that $(S, w) \neq 0$. For example, let $S_a$ be the formal series such that $(S_a, w) = |w|_a$ for each word $w$ where $|w|_a$ denotes the number of occurrences of $a$ in $w$. For example, $(S_a, aba) = 2$ and $(S_a, bb) = 0$. Hence, the support of $S_a$ is the language consisting of all words that contains at least one $a$. As shown in Example 2.1, $S_a$ is a recognizable series. It is well-known that a language is regular if and only if the language is the support of a recognizable series with coefficients taken from natural numbers.

The coefficient of a word $w$ can represent various quantities such as the cost needed for an operation on $w$, the probability that $w$ is emitted from an information source, etc. One of the possible applications of formal series is the string counting in vulnerability analysis of software (also see related work below). Typical vulnerability analysis of a client-side code in web application reduces to counting the strings that are generated by the code and have a pattern abused by an attacker. Finite automata (FA) are a useful tool for the string counting because many operations on strings used in a code can be represented by FA and the class of regular languages has the decidability of basic problems and the closure property on language operations. When we want to conduct a more elaborated analysis such as quantitative information flow (QIF) analysis based on Shannon entropy, the string counting does not suffice but we need to compute the probability that a string is generated by a code. For a program $P$ with a secret input $X$ and an observable output $Y$, QIF of $P$ is defined as $H(X) - H(X|Y)$ where $H(X)$ is the entropy of $X$ (the initial uncertainty of $X$) and $H(X|Y)$ is the conditional entropy of $X$ given $Y$ (the remaining entropy of $X$ after observing $Y$), which is defined based on the probabilities for $X$ and $Y$ [24], [28]. Let us take an example. Assume that an attacker wants to guess the password $X$ of a victim and he knows from $Y$ that a string consisting of only a single character such as ‘aaa’ is not allowed to be a password by the security policy. Counting the possible candidate passwords would suffice for a simple vulnerability analysis. Let $\Sigma = \{a, b\}$ be an alphabet. The constraint for a string $w$ to be a password is represented by the regular expression $r = a^*b\Sigma^* \cup b^*a\Sigma^*$. For example, the number of the possible passwords of length three is six, which can be computed by constructing an FA accepting $L(r)_4 = \{ w \mid w \text{ matches } r \text{ and } |w| = 4 \}$. (Note that $L(r)_d$ is a finite set.) If we want to conduct QIF analysis, we further need to compute the probability of the possible passwords of length $d$. If $p(a) = 2/3$ and $p(b) = 1/3$, the probability for $d = 3$ is $\sum_{|w|=3, w=aaan.bbb} p(w) = 1 - 8/27 - 1/27 = 18/27$. This computation can be done by using a formal series as

Manuscript received August 21, 2017.
Manuscript publicized March 16, 2018.
† The authors are with Nagoya University, Nagoya-shi, 464-0814 Japan.
a) E-mail: trungchubao@sqlab.jp
DOI: 10.1587/transinf.2017FOP0003
follows: First, we construct a recognizable series \( S_e \), such that \( (S_e, w) \) is the probability for \( w \), e.g., \( (S_e, aba) = 4/27 \) and \( (S_e, aaa) = 0 \), by augmenting an FA accepting \( a^* b \Sigma^* \cup b^* a \Sigma^* \) with \( p(a) = 2/3 \) and \( p(b) = 1/3 \). Secondly, we compute the sum of the coefficients of each word \( w \) of length \( d \) in \( S_e \), which will be denoted as \( CC(S_e, d) \) in the sequel.

In this paper, we define counting problems for formal series and propose algorithms for the problems for recognizable and algebraic series. We first define two notions on counting for formal series, the coefficient count of \( S \) with length \( d \) (denoted \( CC(S, d) \)) and the support count of \( S \) with length \( d \) (denoted \( SC(S, d) \)), which are natural but different extensions of the number of words of length \( d \) belonging to a given language. We then show that for a given recognizable series \( S \) and a natural number \( d \), \( CC(S, d) \) can be computed in \( O(T \log d) \) time where \( T \) is an upper-bound of time needed for a single matrix operation, and if the state-transition matrices are commutative, \( SC(S, d) \) can be computed in a polynomial order of \( d \). The set of strings generated by a recursive program is expressed by a context-free grammar (CFG) more precisely than FA. However, the previous study translates the obtained CFG into an FA [8]. Algebraic series is an extension of CFGs in the same sense as recognizable series is an extension of FAs. In this paper, we propose an algorithm that computes \( CC(S, d) \) in square time of \( d \) for an algebraic series \( S \).

There are various data structures other than strings generated in a program, such as trees, linked lists and graphs. For QIF analysis or probabilistic testing of programs that dynamically generate such data structures, counting methods for those data structures have also been studied [11]. Tree series is an extension of formal series from strings to trees, which assigns a value of a semiring to each tree [1]. This paper attempts to extend the proposed notion of \( CC(S, d) \) and \( SC(S, d) \) to trees. Unlike strings, even if the size is determined, trees can take different shapes. We extend the coefficient count problem to tree series, denoted as \( CC(S, \tau) \), as follows: for a given tree series \( S \) and a tree \( \tau \), compute the sum of the coefficients of all the trees in \( S \) that have the same shape as \( \tau \). We extend the support count problem to tree series similarly and propose algorithms for computing \( CC(S, \tau) \) and \( SC(S, \tau) \). For an application to QIF analysis of a program that generates trees, we can combine an algorithm \( A \) for \( CC(S, \tau) \) and a tree enumeration algorithm \( B \) that enumerates all the unlabeled trees of a given size without repetition (e.g., [25]), in such a way that for a given size \( d \), we run \( B \) and for each output from \( B \), we run \( A \) and sum up the counting results.

An overview of a vulnerability analysis based on counting can be illustrated in Fig. 1 where formal model corresponds to formal series in this paper while it corresponds to FA or logical formula in the previous work. This paper focuses on counting (within the dashed line in the figure) and the other steps is out of the scope. To empirically evaluate the efficiency of the proposed algorithms, we show the CPU time to compute \( CC(S, d) \) for some CFGs as \( S \), one of which represents the syntax of C language. Finally, to examine an applicability of the proposed method to vulnerability analysis, we have implemented a prototype counting tool that takes a constraint with a restricted syntax, translates it into a CFG and applies our proposed counting algorithm to the translated CFG. We present the results conducted by the tool for more than 17,000 instances taken from Kaluza Benchmark.

**Related work** String constraints solving has been widely studied for analysis of vulnerabilities such as XSS (Cross Site Scripting) and SQL injection in web applications mainly in a client side. The common basic approach is to verify whether the intersection of the set of strings dynamically generated by web applications (target programs) and the set of instances of attack patterns is empty (safe) or not (vulnerable). Formally, an input of the problem is a set of constraints on string variables. A class of constraints varies depending on whether or not we can use each of a relational constraint (e.g., \( z = xy \) meaning that \( z \) is the concatenation of \( x \) and \( y \)), a regular constraint (e.g., \( z \in a^*bc \), meaning that \( z \) belongs to the regular language \( a^*bc \)), substring, index constrains, etc.

String counting further aims at quantitatively showing the vulnerability (if exists) by counting the strings satisfying given constraints. An efficient counting is important because for analyzing the safety of a target system, it is not sufficient to know the satisfiability of the constraint representing the behavior of the system, because it is required to know how many solutions (or models) of the constraint exist. Furthermore, providing an efficient and precise model counting method is useful not only in string constraint solving but also in quantitative information analysis, which is a promising research area that measures the amount of secure information leaked by observing the output values and the behavior of the system including side channel information [5]. Though the results of this paper may contribute to the latter, we briefly describe the related work in both topics.

**String constraint solving:** There have been two major approaches to string constraints solving: bit-vector encoding for bounded-length strings and automata (or formal grammar) theoretic approach. HAMPI [15] is a constraint solver for bounded-length strings represented in bit vectors obtained from a target program via context-free grammars (CFGs). Kaluza [27] is also a constraint solver for fixed-length strings for symbolic execution of JavaScripts. Though these tools are efficient, the soundness of the analysis is not guaranteed because the length of strings is bounded.

JSA [8] is a static analysis tool for Java programs and it translates a flow graph of Java into a CFG, which is in turn translated into a finite automaton (FA) as an over-

---

**Fig. 1** A vulnerability analysis
approximation. Minamide [23] proposes a method of over-
approximating HTML documents generated by PHP codes
by CFGs. As studies on solving string constraints, [13] pro-
vides an algorithm for subset constraints among regular lan-
guages, and [14] proposes a method for lazily solving con-
straints for regular languages. STRANGER [33] is a
tool for constraints on strings generated by PHP codes. PISA [29]
uses monadic second-order logic as a represenation of reg-
ular languages so that the index (position) in a string can be
explicitly specified. Alkakaf, Bultan, et al. propose an ef-
ficient method of computing an over-approximation of the
fixpoint of recursive constraints for analyzing JavaScript [3]
and PHP [34].

Another approach complementary to the above auto-
maton theoretic methods uses finite transducers, which is an
extension of FA by allowing outputs as well as inputs [32].
BEK [12] uses symbolic finite transducers (SFT) to model a
sanitizer in PHP code. [9] and [19] extend this approach by
extended SFTs in which an input is an arbitrary string not
limited to a single character to represent a decoder, and by
using parameterized array, respectively.

Some of recent studies design and implement a string
constraint solver as an SMT (satisfiability modulo theories)
solver on top of a general SAT solver. Z3-str [35] is a
DPLL(T)-based string constraint solver. It supports equa-
tions, length, substring and replacement as constraints, but
does not support regular language containment (member-
ship). More recent tools such as CVC4 [20], S3 [30] and
Norn [2] support regular language containment. For exam-
ple, Norn is based on [1]; which provides a sound over-
approximation of the strings generated by a Horn-clause
program. It is unknown whether the satisfiability of the
string logic with concatenations and FSTs is decidable or
not (see [1] for discussion). [21] showed the problem is de-
cidable for the straight-line fragment of the logic.

String counting: [22] proposes a string counting algorithm
based on generating functions and reports their counting
tool SMC. Their algorithm decomposes the constraints into
the sub-constraints until a computable closed-form gener-
ating function is available for each sub-constraint. The
weakness is the precision is not good because the intersec-
tion of the set of strings that satisfy the decomposed sub-
constraints is usually large. A model counting algorithm
in [4] is based on the fast calculation of the transfer matrix
of a given regular language as mentioned in Sect. 2 of
this paper. ABC [4] is a counting tool that supports string con-
straints including equation, indexOf, substring, replacement
and regular language containment by inheriting the tech-
niques from [3], [34]. ABC contributes as a string counting
module for QIF analysis of side-channels in [5]. Recently,
S3# [31] greatly improved the precision of the counting for
recursive constraints by introducing a recurrence equation
that represents the counting instead of eliminating a ‘non-
progression’ equation.

Quantitative information flow and model counting: Model
counting including string counting can contribute to quan-
titative information flow analysis, which is the problem of
calculating the number of candidate secret values when an
attacker knows observable values (see [28] as a comprehen-
sive tutorial). There have been two directions for QIF, i.e.,
type-based and model-counting. Type-based approach is ef-
cient but the precision is low. In a model-counting method,
a program is translated into a logical formula (or constraints)
and the formula is fed to a SAT or SMT solver that can
count the models. Due to the space limitation, we review
only two papers related to the topic of this paper. [16] pro-
poses a precise QIF analysis method by reducing the prob-
lem to a model counting of integral points within a given
domain (polyhedra), which enables us to use a symbolic
counting tool such as LattE [18]. [11] proposes an approx-
imated counting methods for complex data structures such
as linked lists dynamically generated in a heap by keeping
the numeric fields in the generated memory cells symbolic.

The rest of the paper is organized as follows. In Sect. 2,
we provide the definition of recognizable series and related
notions. Section 3 introduces two notions for counting with
recognizable series and proposes efficient counting algo-
rithms for them. These notions and results are extended to
recognizable tree series in Sect. 4. Also, we provide a dy-
namic programming algorithm that computes CC(S, d) for
algebraic series S. Section 5 describes our implemented
string counter for context-free grammars and some exper-
imental results. Section 6 concludes the paper.

2. Counting for Recognizable Series

2.1 Formal Series and Recognizable Series

Let A be an alphabet and A∗ be the set of all strings over A.
An element of A is called a letter and an element of A∗ is
called a word over A. Let ε denote the empty word. Let R
be a semiring. A formal series S (over A with coefficients
in R) is a function S : A∗ → R. The image by S of a word
w is denoted by (S, w) and it is called the coefficient of w in
S. The class of formal series over A with coefficients in R
is denoted by R((A)). A formal series S is also written in the
summation notation as follows:

\[ S = \sum_{w \in A^*} (S, w)w. \]

Definition 2.1 (support): For a formal series \( S \in R((A)) \),
the support of S, denoted by supp(S) is defined by

\[ \text{supp}(S) = \{ w \in A^* \mid (S, w) \neq 0 \}. \]

Let I and J be index sets. \( R^{I \times J} \) denotes the set of ma-
trices over R indexed by I × J. If I = \{1, 2, ..., m\} and
J = \{1, 2, ..., n\}, \( R^{I \times J} \) is also written as \( R^{m \times n} \).

Definition 2.2 (recognizable series): A formal series \( S \in
R((A)) \) is recognizable if there is an integer \( n \geq 1 \) and a
morphism of monoids \( \mu : A^* \to \mathbb{R}^{n \times n} \) (i.e., \( \mu \) is the identity matrix \( E \) and \( \mu(w_1 w_2) = (\mu w_1)(\mu w_2) \) for every \( w_1, w_2 \in A^* \)) and two matrices \( \lambda \in \mathbb{R}^{1 \times n} \) and \( \gamma \in \mathbb{R}^{n \times 1} \) such that for all words \( w \),

\[
(S, w) = \lambda(\mu w)\gamma.
\]

The triple \((\lambda, \mu, \gamma)\) is called a linear representation of \( S \) (or \( S \) is represented by \((\lambda, \mu, \gamma)\)), and \( n \) is called the dimension of \((\lambda, \mu, \gamma)\).

**Example 2.1** ([7]): Let \( A = \{a, b\} \) and define \(|w|_a\) as the number of occurrences of the letter \( a \) in \( w \). Let \( \mathbb{N} \) be the semiring of natural numbers. The formal series \( \sum_{a \in A} |w|_aw \) is recognizable. In fact, \( S_a \) is represented by a linear representation \((\lambda, \mu, \gamma)\) where \( \lambda = (1, 0) \), \( \mu a = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \), \( \mu b = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \).

**Proposition 2.1** ([7], [10]): Let \( \mathbb{R} \) be \( \mathbb{N} \) or the Boolean semiring \( \mathbb{B} \). For any recognizable series \( \mathbb{R} \langle \langle A \rangle \rangle \), \( \text{supp}(S) \) is a regular language. Conversely, if \( L \subseteq A^* \) is a regular language, there is a recognizable series \( S \in \mathbb{R} \langle \langle A \rangle \rangle \) such that \( L = \text{supp}(S) \).

**Example 2.2:** Let \( A = \{a, b\} \). \( \mathbb{Z} \) be the semiring of integers, and let \( \mathbb{Z} \langle \langle A \rangle \rangle \) be the series

\[
S_{\text{diff}} = \sum_{a \in A} (|w|_a - |w|_b)w.
\]

A linear representation \((\lambda, \mu, \gamma)\) of \( S_{\text{diff}} \) is defined by

\[
\lambda = (1, 0), \quad \mu a = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mu b = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}.
\]

The support of \( S_{\text{diff}} \) is:

\[
\text{supp}(S_{\text{diff}}) = \{w \in A^* \mid |w|_a \neq |w|_b\},
\]

which is not regular.

Note that a deterministic finite automaton (dfa) \( M = (Q, A, \delta, q_1, F) \) where \( Q = \{q_1, \ldots, q_n\} \) is a finite set of states, \( A \) is an input alphabet, \( \delta : Q \times A \to Q \) is a transition function, \( q_1 \in Q \) is the initial state and \( F \subseteq Q \) is the set of final states, can be regarded as a recognizable series \( S_M \in \mathbb{R} \langle \langle A \rangle \rangle \) where \( \mathbb{R} \) is either \( \mathbb{N} \) or \( \mathbb{B} \), represented by a linear representation \((\lambda, \mu, \gamma)\) such that

- \( \lambda = (1, 0, \ldots, 0) \),
- \( \gamma = \begin{pmatrix} f_1 \\ f_n \end{pmatrix} \) where \( f_i = 1 \) if \( q_i \in F \) and \( f_i = 0 \) otherwise.

In what follows, we identify \( S_M \) with \( M \) and call \( S_M \) a dfa.

**Example 2.3:** Let \( S_{\text{ae}} \) be the recognizable series represented by \((\lambda, \mu, \gamma)\) where

\[
\lambda = (1, 0), \quad \mu a = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mu b = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},
\]

which is the dfa shown in Fig. 2.

![Fig. 2 Dfa S_{ae}](image)

2.2 Counting Problems and Algorithms

For a language \( L \subseteq A^* \), let \( L_d = \{w \in L \mid |w| = d\} \) where \(|w|\) denotes the length of \( w \).

**Definition 2.3** (Coefficient count and support count): Let \( S \) be a formal series over \( A \) and \( d \) be a natural number. Define

\[
CC(S, d) = \sum_{|w| = d} (S, w),
\]

and

\[
SC(S, d) = |\text{supp}(S)_{d}| = |\{w \in A^* \mid (S, w) \neq 0, |w| = d\}|
\]

\[
\Delta(C) = \begin{cases} 1 & \text{if } C \neq 0 \\ 0 & \text{if } C = 0 \end{cases},
\]

which are called the coefficient count of \( S \) with length \( d \) and the support count of \( S \) with length \( d \), respectively.

\( CC(S, d) \) is the summation of the coefficients of words of length \( d \) in \( S \) whereas \( SC(S, d) \) is the number of words of length \( d \) that belong to \( \text{supp}(S) \).

**Theorem 2.1:** For a recognizable series \( S \) over \( A \) represented by \((\lambda, \mu, \gamma)\) and a natural number \( d \in \mathbb{N} \),

\[
CC(S, d) = \lambda \mathbb{P}^d \gamma
\]

where \( \mathbb{P} = \sum_{a \in A} \mu a \). \( CC(S, d) \) can be computed in \( O(\eta \log d) \) time where \( \eta \) is an upper-bound of the time needed for a single matrix operation (addition and multiplication)\(^\dagger\).

**Proof.**

\[
CC(S, d) = \sum_{|w| = d} (S, w) = \sum_{c_i \in A, 1 \leq i \leq c} \lambda(\mu c_1 \cdots \mu c_d)\gamma = \lambda \sum_{a \in A} (\mu a)^d \gamma = \lambda \mathbb{P}^d \gamma.
\]

\(^\dagger\)Note that \( \eta \in O(n^3) \) where \( n \) is the dimension of \((\lambda, \mu, \gamma)\).
Let $d = \sum_{k=0}^L b_k 2^k$ be the binary representation of $d$ where $L = \lfloor \log_2 d \rfloor$ and $b_k = 0$ or $1$ for $0 \leq k \leq L$. Then, we have $\overline{d} = \prod_{b_k = 1, 0 \leq k \leq L} 2^k$, which can be computed within the required time by matrix exponentiation operations. \hfill \Box

A linear representation $(\lambda, \mu, \gamma)$ is commutative if $\mu ab = \mu ba$ holds for every $a,b \in A$.

**Theorem 2.2:** Let $S$ be a recognizable series over $A = \{a_1, \ldots, a_m\}$ represented by a commutative linear representation $(\lambda, \mu, \gamma)$ and $d \in \mathbb{N}$.

\[
SC(S,d) = \sum_{d_1 + \cdots + d_m = d} \left( \begin{array}{c} d \\ d_1, \ldots, d_m \end{array} \right) A[\mu(\alpha_1)^{d_1} \cdots (\mu a_m)^{d_m}] \gamma
\]

where \(\left( \begin{array}{c} d \\ d_1, \ldots, d_m \end{array} \right)\) is the multinomial coefficient of $d = d_1 + \cdots + d_m$ defined as \(\frac{d!}{d_1! \cdots d_m!}\).

$SC(S, d)$ can be computed in $O(d^n (pm + d^2 \log d) \log d)$ time where $\eta$ is an upper-bound mentioned in the previous theorem.

Proof.

\[
SC(S,d) = \sum_{[w] \in A^d} \Delta[(S, w)] = \sum_{[w] \in A^d} \Delta[\lambda(\mu w) \gamma]
\]

\[= \sum_{c \in A, 1 \leq d \leq L} \Delta[\lambda(\mu c^1 \cdots \mu c_d) \gamma]
\]

\[= \sum_{d_1 + \cdots + d_m = d} \left( \begin{array}{c} d \\ d_1, \ldots, d_m \end{array} \right) A[\lambda(\mu a_1)^{d_1} \cdots (\mu a_m)^{d_m}] \gamma.
\]

The number of combinations $d_1, \ldots, d_m$ is $O(d^m)$. In a similar way to the previous theorem, we can show for given $d_1, \ldots, d_m$, it takes $O(pm \log d)$ time to compute $\lambda(\mu a_1)^{d_1} \cdots (\mu a_m)^{d_m}\gamma$, and $O(d^2 \log^2 d)$ time to calculate the multinomial coefficient. \hfill \Box

**Example 2.4:** For $S_a$ in Example 2.1, $\overline{\mu} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$ and $\overline{\mu^2} \overline{\mu} = \begin{pmatrix} 4 & 4 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 12 \\ 0 & 8 \end{pmatrix}$. Therefore, $CC(S_a,3) = (1,0) \begin{pmatrix} 8 & 12 \\ 0 & 8 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 12.$ Since $\mu ab = \mu ba$, we can apply Theorem 2.2. We have

\[
\lambda(\mu a)^i (\mu b)^j \gamma \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i \geq 1 \end{cases}
\]

and so $SC(S_a,3) = 0 + 3 + 3 + 1 = 7.$ \hfill \Box

**Example 2.5:** Let us consider $S_{\text{diff}}$ in Example 2.2. Since $\overline{\mu} = 2E$, $A\overline{\mu}^2 \overline{\gamma} = 2^d A\gamma = 0$ for every $d \geq 1$. Therefore, $CC(S_{\text{diff}}, d) = 0$ for every $d \geq 1$. Since $\mu ab = \mu ba$, we can apply Theorem 2.2. Note that $\lambda(\mu a)^i (\mu b)^j \gamma \geq 1$ if $i \neq j$. Therefore, $SC(S_{\text{diff}},3) = \sum_{k=0}^3 (\frac{1}{2^k}) = (1 + 1)^3 = 8.$ \hfill \Box

**Corollary 2.1:** For a dfa $S$, $SC(S, d) = CC(S, d)$ for every $d \in \mathbb{N}$. $SC(S, d)$ can be computed in $O(\eta \log d)$ time where $\eta$ is an upper-bound mentioned in Theorem 2.1.

Proof. Let $S$ be the dfa represented by $(\lambda, \mu, \gamma)$. By definition, for an arbitrary $w \in A^*$, exactly one component of $\lambda(\mu w)$ is 1 and the other components are all zero. Hence, $(S, w) = 1$ when $(S, w) \neq 0$ and the claim holds. \hfill \Box

As shown above, we obtain the main theorem of [4] as a corollary of Theorem 2.1 although [4] did not explicitly mention an upper-bound of the computational complexity. In fact, [4] uses generating functions for deriving an algorithm that counts the words of a given length in a regular language by multiplication of the “transfer matrix” $\overline{\mu}$. A generating function is a power series over one variable (or a singleton alphabet, say $A = \{z\}$) and can be regarded as a special case of a formal series. A technique common to [4] and this paper is to transform every letter $a \in A$ into an identical letter, say $z$ so that we can merge all the different runs of a dfa (or a recognizable series) for different inputs of the same length into a single run having a weight (a value of the semiring) greater or equal to one, which represents the multiplicity of the runs. This can be done in a simple way by using $\overline{\mu} = \sum_{a \in A} \mu a$ in this paper.

**Example 2.6:** The third example of counting is for $S_{ae}$ in Example 2.3. We have $\overline{\mu} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$. By Theorem 2.1 and Corollary 2.1,

\[
SC(S_{ae}, d) = CC(S_{ae}, d) = (1,0) \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0) \begin{pmatrix} 2^d-1 \\ 2^d-1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2^d-1
\]

for $d \geq 1$ and

$SC(S_{ae},0) = CC(S_{ae},0) = (1,0) \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1.$

By the way,

$S_{ae} = \varepsilon + b + aa + bb + aab + aba + baa + bbb + \cdots .$

By replacing every $a$ and $b$ with $z$ in $S_{ae}$, we obtain the generating function $GF(z)$ of $SC(S_{ae}, d)$:

\[
GF(z) = 1 + z + 2z^2 + 4z^3 + \cdots = 1 + \sum_{k=1}^\infty 2^{k-1} z^k
\]

where we write $1$ instead of $\varepsilon$ according to the usual notation of the unit element in algebra. We have $(GF(z), z^d) = 2^d-1$ for $d \geq 1$ and $(GF(z), 1) = 1$, which coincide with the above results. \hfill \Box
3. Extension to Recognizable Tree Series

Let $F$ be a ranked alphabet where the rank of $f \in F$ is denoted as $r(f)$. Let $F_r = \{ f \mid f \in F, r(f) = r \}$. Define the set of trees over $F$ as the smallest set satisfying:

$f(t_1, \ldots, t_r) \in T(F)$ whenever $t_i \in T(F)$ for $1 \le i \le r$.

For a semiring $R$, a tree series $S$ over $F$ with coefficients in $R$ is a function $S : T(F) \to R$. Similarly to a formal series, the image of a tree $t \in T(F)$ by $S$ is denoted as $(S, t)$ and we write

$$S = \sum_{t \in T(F)} (S, t)t.$$

Let $\mathbb{R} \langle \langle F \rangle \rangle$ denote the class of tree series over $F$ with coefficients in $R$. For a tree series $S \in \mathbb{R} \langle \langle F \rangle \rangle$, the support of $S$ is defined as $\text{supp}(S) = \{ t \in T(F) \mid (S, t) \neq 0 \}$.

We extend recognizability to tree series. A pair $(\mu, \gamma)$ is a representation for $F$ in $\mathbb{R}$ where $Q = \{ 1, \ldots, n \}$ is a finite set of indices (or states), $\mu = \bigcup_{i \geq 0} \mu_i$, with $\mu_i : (\mathbb{R}^Q \times \mathbb{Q})^F$, and $\gamma : \mathbb{R}^Q$. (We will interchangeably write $\mu : F \to \mathbb{R}^Q \times \mathbb{Q}$, $\gamma : \mathbb{R}^Q$, and so on.)

Let $t = f(t_1, \ldots, t_r)$ where $r = r(f)$. We extend $\mu$ to a tree homomorphism in $T(F)$ by:

$$\mu[t] = \sum_{q_1, \ldots, q_r \in Q} \mu_t[q_1] \cdots \mu_t[q_r] \mu(f)[q_1, \ldots, q_r].$$

Also, we define $(\mu \gamma) = \sum_{q \in Q} \mu[q] \gamma[q]$.

**Definition 3.1** (recognizable tree series [10]): A tree series $S \in \mathbb{R} \langle \langle F \rangle \rangle$ is recognizable if there is a representation $(\mu, \gamma)$ such that for all trees $t$,

$$(S, t) = (\mu \gamma)$$

We call $(\mu, \gamma)$ a representation of $S$ (or $S$ is represented by $(\mu, \gamma)$), and $|Q|$ is called the dimension of $(\mu, \gamma)$.

**Example 3.1:** Let $F = \{ f, g, a, b \}$ with $r(f) = r(g) = 2$, $r(a) = r(b) = 0$ and let $S_{4 \times 4 \times 2} \in \mathbb{Z} \langle \langle F \rangle \rangle$ be a recognizable tree series represented by $(\mu, \gamma)$ where

- the index set is $Q = \{ 1, 2 \}$,
- $\mu_{0a} = (1, 1), \mu_{0b} = (1, -1)$,
- $\mu_2 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ and $\mu_2 g = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$,
- $\gamma = \begin{pmatrix} 0 & 1 \\ 1 \end{pmatrix}$.

For example,

- $\mu(a, b)[1] = 1 \cdot 1 + 1 \cdot (-1) \cdot 0 + 1 \cdot 1 \cdot 0 = 1$,
- $\mu(a, b)[2] = 1 \cdot 1 + 1 \cdot (-1) + 1 \cdot 1 = 1$,
- $\mu_f(a, b) = 1 \cdot 1 + 1 \cdot (-1) + 1 \cdot 1 = 1$,
- $\mu_f(a, b) = (1, -1)$.

In general, $(S_{4 \times 4 \times 2}, t) = (|t|_a + |t|_b) - (|t|_a + |t|_b)$. □

As recognizable series can be seen as an extension of dfa, deterministic bottom-up tree automata can be regarded as a special case of recognizable tree series. A deterministic bottom-up tree automaton (dbta) is a tuple $M = (Q, F, R, Q_0)$ where $Q$ is a finite set of states, $F$ is a ranked alphabet, $R$ is a finite set of transition rules satisfying the conditions described below, $Q_F \subseteq Q$ is the set of final states. A transition rule in $R$ has the shape $f(q_1, \ldots, q_r) \to q$ where $q_1, \ldots, q_r, q \in Q$. For every $f \in F$, $q_1, \ldots, q_r, q \in Q$, there is exactly one rule in $R$ whose left-hand side is $f(q_1, \ldots, q_r)$ (i.e., we assume $M$ is complete without loss of generality). $M$ can be regarded as a recognizable tree series represented by $(\mu, \gamma)$ where

- $\mu_i(f)[q_1, \ldots, q_r, q] = \begin{cases} 1 & \text{if } f(q_1, \ldots, q_r) \to q \in R, \\ 0 & \text{otherwise,} \end{cases}$
- $\gamma(q) = 1$ if $q \in Q_F$.

We define the equivalence relation $\approx$ over $T(F)$ as the smallest relation satisfying: for every $f, g \in F$ with $r(f) = r(g)$, $f(q_1, \ldots, q_r) \equiv g(q_1, \ldots, q_r)$ whenever $q_i \equiv t_i (1 \le i \le r)$. If $s \approx t$, we say $s$ is shape equivalent with $t$.

Similarly to the coefficient and support counts of formal series, we will define two countings of a given tree series with respect to the shape equivalence with a given tree, instead of the length of words.

**Definition 3.2** (coefficient count and support count): Let $L \subseteq T(F)$ be a tree language over $F$ and $\tau$ be a tree in $T(F)$. Define

$$CC(S, \tau) = \sum_{t \approx \tau} (S, t),$$

and

$$SC(S, \tau) = |\text{supp}(S)_{\approx \tau}| = |\{ t \in T(F) \mid (S, t) \neq 0, t \approx \tau \}|$$

which are called the coefficient count of $S$ with shape $\tau$ and the support count of $S$ with shape $\tau$, respectively. □

We do not yet know an efficient computing method similar to matrix exponentiation in recognizable series. The following theorem provides an algorithm that utilizes the summation $\sum_{f \in F} \mu_r(f)$ as the state transition (instead of tracing an independent transition $\mu_r(f)$ and summing up), and computes $CC(S, \tau)$ recursively on the structure of $\tau$.

**Theorem 3.1**: For a recognizable tree series $S$ over $F$ represented by $(\mu, \gamma)$ and a tree $\tau \in T(F), \ldots$
where $\overline{\mu} = \bigcup_{\tau \geq 0} \overline{\mu}_{\tau}$, $\overline{\mu}_{\tau}(\varphi) = \sum_{f \in F_{\tau}} \mu_{\tau}(f)$ for any $\varphi \in F_{\tau}$.

$CC(S, \tau) = (\overline{\mu}_\tau)\gamma$

Proof. Let $\tau$ be an upper-bound of the time needed for computing $v_1 \cdots v_{\mu}(f)[q_1, \ldots, q_r], q$ for given $v_1 \in \mathbb{R}$, $q_i \in Q$ ($1 \leq i \leq r$) and $q \in Q$.

- $n$ is the dimension of $(\mu, \gamma)$,
- $\rho$ is the maximum rank of $f \in F$, and
- $|\tau|$ is the size of $\tau$ defined by $|\varphi(t_1, \ldots, t_r)| = 1 + \sum_{i=1}^{|\tau|}$.

We will show that for any $\tau \in T(F)$, $\overline{\mu}_\tau = (2^{\lceil |\tau| \rceil}, 0)$ by the structural induction on $\tau$.

The claim holds in the basis case because $\overline{\mu}_{0}a = \overline{\mu}_{0}b = (2, 0)$. Let $\tau = \varphi(t_1, t_2) \in T(F)$, and assume that the claim holds for $t_1$ and $t_2$. Then, $\overline{\mu}_\varphi(t_1, t_2)(1) = \overline{\mu}_{t_1}(1) \times \overline{\mu}_{t_2}(1) \times 2 = 2^{\lceil |\tau| \rceil} \cdot 2$. Therefore, $\overline{\mu}_\varphi(t_1, t_2)\varphi(1) \times \overline{\mu}_{t_2}(1) \times 2 = 2$. Namely, $\overline{\mu}_\varphi(t_1, t_2) = (2^{\lceil |\tau| \rceil}, 0)$ and the claim holds in the inductive case. Thus, the claim holds by the induction. By Theorem 3.1, $CC(S_{\text{diff}}, \tau) = (2^{\lceil |\tau| \rceil}, 0)$ holds and the claim follows.

For $SC(S, \tau)$, we even do not know a way of merging runs of $S$ on different trees that have the same shape $\tau$. An upper-bound of the time to compute $SC(S, \tau)$ is given in the next theorem.

**Theorem 3.2:** Let $S$ be a recognizable tree series over $F$ represented by $(\mu, \gamma)$ and let $\tau \in T(F)$. $SC(S, \tau)$ can be computed in $O(\xi n^{r+1} |\tau| r^{|\tau|})$ time where $\rho$ is the maximum rank of $f$ in $F$.

If $S$ is a dbta, $SC(S, \tau)$ coincides with $CC(S, \tau)$, which is similar to the fact that $SC(S, d) = CC(S, d)$ for a dfa $S$, and we can use the algorithm in Theorem 3.1 to compute $SC(S, \tau)$.

**Corollary 3.1:** For a dbta $S$ over $F$, $SC(S, \tau) = CC(S, \tau) = (\overline{\mu}_\tau)\gamma$ for every $\tau \in T(F)$.

Proof. We can show that for every $\tau \in T(F)$, there is exactly one $q \in Q$ such that $\mu_{\tau}(q) = 1$ and $\mu_{\tau}(q') = 0$ for every $q' \neq q$. Hence, $(S, \tau) = \Delta((S, \tau))$ holds and the claim follows.

### 4. Counting for Algebraic Series

In this section, we consider counting problems for algebraic series. While counterparts of recognizable series in formal language theory are regular languages, counterparts of algebraic series are context-free languages. We first introduce definitions and well-known properties of algebraic series based on [10], [17] and then present an algorithm that computes $CC(S, d)$, the coefficient count of a given algebraic series $S$ by dynamic programming. Computing coefficients of an algebraic series has applications to static analysis of programs because the behavior of a recursive program can often be modeled by a context-free language and the result of static analysis such as dataflow analysis can be represented as a coefficient of the corresponding algebraic series (see [26] for example).

A polynomial $S \in \mathbb{R}(\langle A \rangle)$ is a formal series such that $\text{supp}(S)$ is finite. The class of all polynomials over $A$ with coefficients in $\mathbb{R}$ is denoted as $\mathbb{R}(A)$.

We fix an alphabet $A$ and a semiring $\mathbb{R}$ and let $Y$ be a countable set of variables disjoint with $A$.

**Definition 4.1** (algebraic system): An $(\mathbb{R})$-algebraic system is a finite set of equations of the form
Definition 4.2: A solution of a given algebraic system consists of $n$ formal series $R_1, \ldots, R_n \in \mathbb{R} \langle \mathcal{A} \rangle$ satisfying the system in the sense that when each component $y_i (1 \leq i \leq n)$ is replaced with $R_i$, $y_i$ becomes equal to $p_i$ for every $i (1 \leq i \leq n)$. For a vector $v$, let $v^T$ denote the transpose of $v$. Formally, let $R = (R_1, \ldots, R_n)^T \in \mathbb{R} \langle \mathcal{A} \rangle^n$ be an $n$-column vector of formal series and define the morphism $h_R : (A \cup Y)^* \rightarrow \mathbb{R} \langle \mathcal{A} \rangle$ by $h_R(y_i) = R_i (1 \leq i \leq n)$ and $h_R(a) = a$ for $a \in A$. Furthermore, extend $h_R$ to a polynomial $p \in \mathbb{R} \langle A \cup Y \rangle$ by

$$h_R(p) = \sum_{u \in (A \cup Y)^*} (p, w) h_R(w).$$

$R$ is a solution of the algebraic system $y_i = p_i (1 \leq i \leq n)$ if $R_i = h_R(p_i)$ for all $i (1 \leq i \leq n)$.

We say $S \in \mathbb{R} \langle \mathcal{A} \rangle$ is quasi-regular if $(S, e) = 0$. It is well-known that every proper algebraic system $y_i = p_i (1 \leq i \leq n)$ has a unique solution $R = (R_1, \ldots, R_n)^T$ such that every component $R_i (1 \leq i \leq n)$ is quasi-regular. $R$ is called the strong solution of the system. The strong solution of a proper algebraic system $y_i = p_i (1 \leq i \leq n)$ can be given by the limit $R = \lim_{j \rightarrow \infty} R^j$ of the sequence $R^j$ defined by $R^0 = (0, \ldots, 0)^T$ and $R^{j+1} = (h_R(p_1), \ldots, h_R(p_n))^T$ for $j \geq 0$.

Example 4.1: (1) For an $\mathbb{N}$-algebraic system $y = ayb + ab, R^0 = 0, R^1 = ab, R^2 = ab + a^2b^2, \ldots,$ and $R = \sum_{n \geq 1} a^n b^n$.

(2) For an $\mathbb{N}$-algebraic system $y = ay + a, R^0 = 0, R^1 = a, R^2 = a^2, R^3 = a + (a + a^2)(a + a^2 + a^3 + a^4), \ldots, R = \sum_{i=1}^{n} C_n \cdot a^{i}$ where $C_n$ is the $n$-th Catalan number: $C_n = \frac{1}{n+1} \binom{2n}{n}$.

Definition 4.2 (algebraic series): A formal series $S \in \mathbb{R} \langle \mathcal{A} \rangle$ is $\mathbb{R}$-algebraic if $S = (S, e) + S'$ where $S'$ is some component of the strong solution of a proper $\mathbb{R}$-algebraic system. The class of $\mathbb{R}$-algebraic series over $A$ is denoted by $\mathbb{R} \langle \mathcal{A} \rangle$.

A language $L \subseteq A^*$ is $\mathbb{R}$-algebraic if $L = \text{supp}(S)$ for some $S \in \mathbb{R} \langle \mathcal{A} \rangle$.

A context-free grammar (CFG) is a tuple $(N, T, P, S)$ where $N$ and $T$ are finite sets of nonterminal and terminal symbols, respectively, $P$ is a finite set of (production) rules in the form $A \rightarrow \gamma$ where $A \in N$ and $\gamma \in (N \cup T)^*$ and $S \in N$ is the start symbol. The derivation relation $\Rightarrow_G$ is defined in the usual way. For $A \in N$, let $L_G(A) = \{ w \in T^* | A \Rightarrow_G w \}$. The context-free language generated by $G$ is $L(G) = L_G(S)$.

Proposition 4.1 ([10], [17]): A language $L$ is context-free if and only if $L$ is $\mathbb{N}$-algebraic if and only if $L$ is $\mathbb{R}$-algebraic. Proof sketch. For a given algebraic system $y_i = p_i (1 \leq i \leq n)$, construct CFG $G = (\{y_1, \ldots, y_n\}, A, P, y_1)$ where $y_1 \rightarrow w \in P$ iff $(p_i, w) \neq 0$. Conversely, for a given CFG $G = (\{y_1, \ldots, y_n\}, A, P, y_1)$, construct the algebraic system $y_i = p_i (1 \leq i \leq n)$ where $(p_i, w) = 1$ if $y_i \rightarrow w \in P$ and $(p_i, w) = 0$ otherwise.

Definition 4.3 (Chomsky normal form): An $\mathbb{R}$-algebraic system $y_i = p_i (1 \leq i \leq n)$ is in Chomsky normal form if $\text{supp}(p_i) \subseteq A \cup Y^2$ for every $i (1 \leq i \leq n)$.

As is the case of context-free languages, for a proper $\mathbb{R}$-algebraic system $\alpha : y_i = p_i (1 \leq i \leq n)$, we can construct an $\mathbb{R}$-algebraic system $\beta$ in Chomsky normal form such that every component of the strong solution of $\alpha$ is a component of the strong solution of $\beta$. Hence, we assume without loss of generality that a given proper $\mathbb{R}$-algebraic system is in Chomsky normal form.

We provide an algorithm that computes $CC(S, d)$ for an algebraic series $S$ and $d \in \mathbb{N}$.

**Theorem 4.1:** Let $\alpha : y_i = p_i (1 \leq i \leq n)$ be an $\mathbb{R}$-algebraic system in Chomsky normal form and let $R = (R_1, \ldots, R_n)^T$ be the strong solution of $\alpha$. Then, for each $i (1 \leq i \leq n)$ and $d \in \mathbb{N}$,

$$CC(R_i, d) = \sum_{y_j y_m \in \text{supp}(p_i) \cap Y^2} CC(R_j, k)CC(R_m, d-k) + \sum_{a \in \text{supp}(p_i) \cap A} (p_i, a)\Delta[d = 1].$$

Moreover, $CC(R_i, d)$ can be computed in $O(\xi d^2)$ time where $\xi = \max_{1 \leq i \leq n} |p_i|$.

**Proof.** By definition of strong solution, for every $i (1 \leq i \leq n)$,

$$R_i = h_R(p_i) = \sum_{u \in (A \cup Y)^*} (p_i, w) h_R(w)$$

$$= \sum_{y_j y_m \in \text{supp}(p_i) \cap Y^2} (p_i, y_j y_m) R_j R_m + \sum_{a \in \text{supp}(p_i) \cap A} (p_i, a)\Delta[w = a].$$

Therefore, for every $i (1 \leq i \leq n)$,

$$CC(R_i, d) = \sum_{|u| = d} (R_i, u)$$

$$= \sum_{y_j y_m \in \text{supp}(p_i) \cap Y^2} (R_j, u)(R_m, v) + \sum_{a \in \text{supp}(p_i) \cap A} (p_i, a)\sum_{|w| = d} \Delta[w = a].$$

CC(Ri, d) can be computed simultaneously for all i (1 ≤ i ≤ n) in O(ξd^2) time by dynamic programming where ξ = n max_{1≤i≤n} |p_i|.

We can extend the above algorithm to apply to an arbitrary algebraic series.

Example 4.2: Consider the following algebraic system:

\[
y_1 = y_2 y_3 + y_3 y_5, \quad y_2 = ay_2 + a, \quad y_3 = by_3 + bc, \quad y_4 = ay_4b + ab, \quad y_5 = cy_5 + c.
\]

Let \( R = (R_1, \ldots, R_3)^T \) be the strong solution of the system. CC(Ri, d) (1 ≤ i ≤ 5, 1 ≤ d ≤ 5) can be computed by Theorem 4.1 as shown below.

<table>
<thead>
<tr>
<th>R_1</th>
<th>R_2</th>
<th>R_3</th>
<th>R_4</th>
<th>R_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Also,

\[
CC(R_1, 6) = \sum_{k=1}^{5} (CC(R_2, k)CC(R_3, 6 - k) + CC(R_4, k)CC(R_5, 6 - k)) = 4.
\]

R_1 = \sum_{i\not=j, i,j\geq1} (a_ib^ic^j + a^2b^3c^2) + \sum_{i\geq2} 2a^2d^3c^2 and supp(R_1) = \{a^ib^jc^k | i, j \geq 1\} \cup \{a^2b^3c^2 | i, j \geq 1\} is an (inherently) ambiguous context-free language.

For a given CFG \( G \), let \( \alpha_G : y_i = p_i (1 \leq i \leq n) \) be the algebraic system constructed in the proof of Proposition 4.1. Let \( R = (R_1, \ldots, R_n)^T \) be the strong solution of \( \alpha_G \). By the construction of \( \alpha_G \), \( (R_1, w) \) represents the number of different derivation trees of \( w \) in \( G \). Hence, we have the following corollary similar to Corollary 2.1, which states that \( SC(S, d) = CC(S, d) \) and thus \( SC(S, d) \) can be computed in square time of \( d \) if an algebraic series \( S \) corresponds to an unambiguous CFG.

Corollary 4.1: Let \( G \) be a CFG and \( \alpha_G \) be the algebraic system constructed in the proof of Proposition 4.1. Let \( R = (R_1, \ldots, R_n)^T \) be the strong solution of \( \alpha_G \). If \( G \) is unambiguous, \( |L(G)_d| = SC(R_1, d) = CC(R_1, d) \).

5. Experimental Evaluation of the Algorithm for Algebraic Series

To evaluate the efficiency and applicability of the algorithm proposed in Sect. 4, we conducted the following experiments.\(^\dagger\) Based on Theorem 4.1, we implemented an algorithm for solving the following string counting problem for CFGs and conducted experiments for a few CFGs (Sect. 5.1).

\( G \) represents the syntax of C programming language. The size of \( G \) is shown in Table 1.

<table>
<thead>
<tr>
<th>No. of terminal symbols</th>
<th>151</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of non-terminal symbols</td>
<td>143</td>
</tr>
<tr>
<td>No. of rules</td>
<td>431</td>
</tr>
</tbody>
</table>

Next, on top of the above implementation, we prototyped a counting tool for string constraints (Sect. 5.2). The tool was implemented in C++ and the experiments were conducted in the environment: core i7-6500U, CPU@2.5GHz x 4, 8GB RAM, Ubuntu 16.10 64bits. GMP library is utilized to deal with big numbers.

5.1 Basic Performance

We first show the experimental results on two CFGs \( G_1 \) and \( G_2 \). Both of them are taken from [6], with minor modification on \( G_1 \) to make it fit to be input into our implementation. Then, we empirically estimate the time complexity of the algorithm and compare it with the result given in Theorem 4.1. Finally, we discuss the counting precision by using another simple CFG.

\( G_1 \) is an unambiguous CFG that specifies a simplified syntax of English language, consisting of 16 rules. The experimental results on \( G_2 \) is shown in Table 3. We fit the result to \( t = \xi d^a \) and we obtain \( a = 2.472 \).

There are two questions about the estimated values of \( a \) for the two experiments. (1) Besides the length bound, on what factors does the running time of counting algorithm depend? (2) Why are the values of \( a \) estimated from the experimental results larger than the theoretical result \( a = 2 \)? For the first question (1), the empirical results show that beside length bound, the size of the input CFG also affects the running time. It took about 3 seconds to count for \( G_1 \) at the length bound of 20 but the same amount of time is sufficient to count at length bound of 400 for \( G_2 \) whose size is roughly 20 times smaller than \( G_1 \)’s. The answer for the second question (2) is that Theorem 4.1 assumes Chomsky

\( \dagger \) We can keep the time complexity \( O(\xi d^2) \) through this extension by revising the algorithm in such a way that an equation of length three or more is virtually divided into equations of length two on the fly.

\( \dagger \dagger \) We also conducted experiments for the algorithms for recognizable series, which will be reported elsewhere.
Normal Form whilst $G_1$ and $G_2$ are not. Our implementation uses a naive extension of the algorithm in Theorem 4.1 so that the time complexity becomes $(q d^m)$ where $m$ is the maximum length of the right-hand sides of the rules\footnote{It is possible to explicitly transform a given CFG to a Chomsky normal form or revise the algorithm so that the given CFG is processed as pointed out in the footnote after Theorem 4.1.}. This made the value of $a$ slightly larger than 2.

Lastly, we discuss the counting precision. Remember that $CC(S, d)$ is the summation of the coefficients of words of length $d$ in $S$ and under the interpretation of a CFG as an algebraic series shown in the proof of Proposition 4.1, the coefficients of a word $w$ corresponds to the number of derivation trees of $w$. Consequently, while our implementation gives the exact answer to the counting problem for an unambiguous CFG, it only outputs an upper bound of the answer for an ambiguous CFG. Let’s consider the following ambiguous CFG [6].

\[
\begin{align*}
S & \rightarrow A.B | a \\
A & \rightarrow S.B | b \\
B & \rightarrow B.A | a
\end{align*}
\]

Table 4 shows the exact count and the output of our implementation. As seen in the table, the precision goes down rapidly when the length bound increases.

5.2 Applicability to String Counting

In this subsection, we give the experimental results conducted by our prototype counting tool and compare it to the state-of-the-art string counter ABC on Kaluza benchmark [4], [22].

Figure 3 shows the syntax of string constraints that our tool accepts as input. This tool cannot handle string operations indexOf, substring, replace, regex (regular expression). To process constraints accepted by this syntax we made some simple extension on the algorithm in Theorem 4.1 about constraints related to $\text{const-string}$ and string length with $\text{const-integer}$ (lines 2, 3 in the Fig. 3). For instance, $\text{len}(A) \neq 3$ is interpreted to $CC(A, 3) = 0$ instead of calculating the right hand side of expression (1) in Theorem 4.1. The tool was implemented on top of the implementation in Sect. 5.1 and, as noted there, the tool is not guaranteed to precisely work for all the constraints accepted by the syntax.

Kaluza benchmark consists of two parts, small and big. The former includes 17,554 files and the latter does 1,342 files. Almost all of files in Kaluza small benchmark satisfy the syntax of our tool with simple equivalence transformation. In fact, there are some regular expression constraints of the form $x \text{in/regex}$ in the benchmark, but almost all of the regex represent only a constant string $c$ and thus they can be converted into $x=c$. Our tool cannot handle precisely the conjunction of rules of CFGs (called CF constraints) because context-free languages (CFLs) are not closed by the intersection. 1,019 out of 1,342 files in Kaluza big include the conjunction of CF constrains rooted at the queried variable, while those in Kaluza small are only 163 out of 17,554 files. Taking this fact into consideration, we compared our tool with ABC on Kaluza small.

We cloned ABC source code and benchmark from: https://github.com/vlab-cs-ucsb/ABC\footnote{There were several branches without mentioning about a stable release in the repository. So, we cloned the master branch as of 2017/12/27}. We ran both ABC and our tool at length bounds 1, 2 and 3 so that we can manually confirm the counts. There was no big

<table>
<thead>
<tr>
<th>Bound</th>
<th>CPU Time (ms)</th>
<th>Count (approx.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3344.53</td>
<td>$2.6 \times 10^3$</td>
</tr>
<tr>
<td>15</td>
<td>3797.49</td>
<td>$1.1 \times 10^3$</td>
</tr>
<tr>
<td>20</td>
<td>3476.46</td>
<td>$6.3 \times 10^2$</td>
</tr>
<tr>
<td>30</td>
<td>3971.00</td>
<td>$3.2 \times 10^2$</td>
</tr>
<tr>
<td>40</td>
<td>4838.96</td>
<td>$2.0 \times 10^1$</td>
</tr>
<tr>
<td>60</td>
<td>11969.30</td>
<td>$1.1 \times 10^1$</td>
</tr>
<tr>
<td>80</td>
<td>45097.40</td>
<td>$7.6 \times 10^0$</td>
</tr>
<tr>
<td>120</td>
<td>421563.00</td>
<td>$4.6 \times 10^{-1}$</td>
</tr>
<tr>
<td>160</td>
<td>2320180.00</td>
<td>$3.3 \times 10^{-2}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bound</th>
<th>CPU Time (ms)</th>
<th>Count (approx.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>3655.17</td>
<td>$3.3 \times 10^3$</td>
</tr>
<tr>
<td>600</td>
<td>5107.08</td>
<td>$8.3 \times 10^3$</td>
</tr>
<tr>
<td>800</td>
<td>9002.97</td>
<td>$2.4 \times 10^4$</td>
</tr>
<tr>
<td>1200</td>
<td>21057.50</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td>1600</td>
<td>47927.00</td>
<td>$1.3 \times 10^5$</td>
</tr>
<tr>
<td>2400</td>
<td>161622.00</td>
<td>$7.3 \times 10^5$</td>
</tr>
<tr>
<td>3200</td>
<td>393007.00</td>
<td>$4.0 \times 10^6$</td>
</tr>
<tr>
<td>4800</td>
<td>1360400.00</td>
<td>$1.2 \times 10^7$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length</th>
<th>Exact Count</th>
<th>Prototype’s Output</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>0.80</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>14</td>
<td>0.57</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>42</td>
<td>0.38</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>132</td>
<td>0.24</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>429</td>
<td>0.15</td>
</tr>
</tbody>
</table>
difference about running time between the tools noticed. The comparison results were consistent through different length bounds (1, 2 and 3) in which ABC and our tool give the same results on 17,183 files. All of those files did not include the conjunction of CF constraints. We manually investigated the other 371 files and found that they can be classified into 21 groups such that all constraints in a same group are very similar. Among those 21 groups, our tool gave correct answers for 18 groups, more precise than ABC for 1 group. The last group included the conjunction of CF constraints which our tool could not accept. 163 files (distributed into 1 group in Ours right, 2 groups in Ours better) among those 371 files included the conjunction of CF constraints. The reason why our tool could precisely answer for 18 groups was that those conjuncts were the repetition of same constraints. Overall, the experimental result shows that our counting algorithm is potentially applicable to a qualified benchmark and it is promising to extend the algorithm.

6. Conclusion

In this paper, we define two notions on counting for formal series, the coefficient count $CC(S, d)$ and the support count $SC(S, d)$. We show that for a given recognizable series $S$ and a natural number $d$, $CC(S, d)$ can be computed in $O(\eta \log d)$ time where $\eta$ is an upper-bound of time needed for a single state-transition matrix operation, and if the state-transition matrices of $S$ are commutative for multiplication, $SC(S, d)$ can be computed in polynomial time of $d$. We also extend the notions to tree series and algebraic series and show that $CC(S, d)$ can be computed in square time of $d$ for an algebraic series $S$.

We implemented a prototype to confirm the efficiency and applicability of the counting algorithm for algebraic series. The time complexity of the algorithm follows the theoretical results. Our prototype’s applicability and its potential was verified on a well-known benchmark.

Though the algorithm gives exact answers for unambiguous CFGs, it only calculates loose upper bound for ambiguous ones. We leave improving this shortcoming one of future work. As mentioned in Sect. 4, our method should work in a similar way for other algebraic systems, so to extend towards arbitrary ones is also a possible improvement. Besides, another direction is to utilize this counter for quantifying information leakage of a program, once have it represented as a CFG [8].

## Acknowledgements

This work was supported by JSPS KAKENHI Grant Numbers JP26540025, JP15H02684.

## References

Bao Trung Chu received his Master’s degree from Graduate School of Information Science, Nara Institute of Science and Technology in 2014. From 2014 to 2015 he had worked as a system engineer in oRo Co., Ltd. Since Apr. 2016, he has pursued his Ph.D. degree in Nagoya University. His field of interest include formal approach to automatic software verification and information security.

Kenji Hashimoto received the Ph.D. degree in Information and Computer Sciences from Osaka University in 2009. From 2009 to 2013, he was an Assistant Professor of Nara Institute of Science and Technology. Since Oct. 2013, he has been an Assistant Professor in Nagoya University. His research interests include formal language, database theory, and information security.

Hiroyuki Seki received his Ph.D. degree from Osaka University in 1987. He was an Assistant Professor, and later, an Associate Professor in Osaka University from 1987 to 1994. In 1994, he joined Nara Institute of Science and Technology, where he was a Professor during 1996 to 2013. Currently, he is a Professor in Nagoya University. His current research interests include formal language theory and formal approach to software development.