Math. Struct. in Comp. Science (2018), *vol.* 28, *pp.* 1415–1450. © Cambridge University Press 2018
doi:10.1017/S0960129518000208 First published online 2 August 2018

Linear pattern matching of compressed terms and polynomial rewriting

MANFRED SCHMIDT-SCHAUß

Institut für Informatik, Fachbereich Informatik und Mathematik, Johann Wolfgang Goethe-Universität, Postfach 11 19 32, D-60054 Frankfurt, Germany
Email: schauss@cs.uni-frankfurt.de

Received 1 February 2012; revised 1 November 2017

We consider term rewriting under sharing in the form of compression by singleton tree grammars (STG), which is more general than the term dags. Algorithms for the subtasks of rewriting are analysed: finding a redex for rewriting by locating a position for a match, performing a rewrite step by constructing the compressed result and executing a sequence of rewrite steps. The first main result is that locating a match of a linear term s in another term t can be performed in polynomial time if s, t are both STG-compressed. This generalizes results on matching of STG-compressed terms, matching of straight-line-program-compressed strings with character-variables, where every variable occurs at most once, and on fully compressed matching of strings. Also, for the case where s is directed-acyclic-graph (DAG)-compressed, it is shown that submatching can be performed in polynomial time. The general case of compressed submatching can be computed in non-deterministic polynomial time, and an algorithm is described that may be exponential in the worst case, its complexity is $n^{O(k)}$, where k is the number of variables with double occurrences in s and n is the size of the input. The second main result is that in case there is an oracle for the redex position, a sequence of m parallel or single-step rewriting steps under STG-compression can be performed in polynomial time. This generalizes results on DAG-compressed rewriting sequences. Combining these results implies that for an STG-compressed term rewrite system with left-linear rules, m parallel or single-step term rewrite steps can be performed in polynomial time in the input size n and m.

1. Introduction

An important concept in various areas of computer science like automated deduction, first-order logic, term rewriting, type checking, are terms (ranked trees), and also terms containing variables (see, e.g., Baader and Nipkow 1998). The basic and widely used algorithms in these areas are matching, unification, term rewriting, equational deduction and asf. For example, a term $f(g(a, b), c)$ may be rewritten into $f(g(b, a), c)$ by the commutativity axiom $g(x, y) = g(y, x)$ for g. Since implemented systems often deal with large terms, perhaps generated ones, it is of high interest to look for compression mechanisms for terms and, consequently, also investigate variants of the known algorithms that also perform efficiently on the compressed terms without prior decompression.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The device of straight line programs (SLP) for the compression of strings is a general one and allows analyses (Rytter 2004) of correctness and complexity of algorithms, see the overview (Lohrey 2012). SLPs are polynomially equivalent to the LZ77-variant of Lempel–Ziv compression (Ziv and Lempel 1977). SLPs are non-cyclic context free grammars (CFGs), where every non-terminal has exactly one production in the CFG, such that any non-terminal represents exactly one string. Basic algorithms are the equality check of two compressed strings, which requires polynomial time (Plandowski 1994) (see Lifshits 2007 for an efficient version and Jez 2012 for a proposal of a further improvement), and the compressed pattern match, i.e. given two SLP-compressed strings s, t, the question whether s is a substring of t can also be solved in polynomial time in the size of the SLPs.

A generalization of SLPs for the compression of terms are singleton tree grammars (STG) (Gascón *et al.* 2008; Levy *et al.* 2006, 2011; Schmidt-Schauß 2005), a specialization of straight line context free (SLCF) tree grammars (Busatto *et al.* 2005, 2008; Lohrey *et al.* 2009, 2012), where linear SLCF tree grammars are polynomially equivalent to STGs (Lohrey *et al.* 2009). Basic notions for tree grammars and tree automata can be found in Comon *et al.* (1997). Besides using the well-known node sharing, also partial subtrees (contexts) can be shared in the compression. The Plandowski–Lifshits equality test of non-terminals can be generalized to STGs and requires polynomial time (Busatto *et al.* 2005; Schmidt-Schauß 2005) in the size of the STG.

A naive generalization of the pattern match is to find a compressed ground term in another compressed ground term, which can be solved by translating this problem into a pattern match of compressed pre-order traversals of the terms. The interesting generalization of the pattern match is the following submatching problem (also called encompassment): given two (STG-compressed) terms s, t, where s may contain variables, is there an occurrence of an instance of s in t? A special case is matching, where the question is whether there is a substitution σ, such that $\sigma(s) = t$, which is shown to be in PTIME in Gascón *et al.* (2008, 2011), including the computation of the (unique) compressed substitution. Other related works are Comon (1995), Salzer (1992) and Hermann and Galbavý (1997) on term schematizations that investigate a form of compression as well as representing infinite sets of terms, and related algorithms.

In this paper, we describe algorithms for answering the submatching question, and which only operate on the STGs. We show that if s is STG-compressed and linear, then submatching can be solved in polynomial time (Theorem 3.17). In the case that s is ground and compressed or that s is directed-acyclic-graph (DAG)-compressed, we describe less complex algorithms that solve the submatching question in polynomial time (Theorems 4.2 and 4.8). In the general case, we describe a non-deterministic algorithm that runs in polynomial time, or making it deterministic, an algorithm that runs in time $O(n^{c|FVmult(s)|})$ (Theorem 5.2), where n is the size of the STG, and $|FVmult(s)|$ is the number of variables that occur more than once in s. This is an exponential-time algorithm, but in a well-behaved parameter: If the number of multiply occurring variables in s is bounded by k, then the matching algorithm runs in polynomial time (in the input size). In Theorem 5.4, it is shown that in case the number of occurrences of variables is small, then submatching under STG-compression can be computed in polynomial time.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

As an application and an easy consequence of the submatching algorithms, a single (parallel or single-position) deduction step on compressed terms by a compressed left-linear rule can be performed in polynomial time. We also show that a sequence of n rewrites with a left-linear term rewriting system (TRS) can be performed in polynomial time, where the TRS as well as the to-be-reduced term are compressed by STGs (see Theorem 6.4). Our result confirms results on complexity of rewrite derivations under DAG-compression (Avanzini and Moser 2010), namely that rewrite systems with a polynomial runtime complexity can be implemented such that the algorithm requires polynomial time. Another potential application is a querying mechanism for compressed XML databases (Lohrey *et al.* 2010).

An introductory example illustrating the compression and equational deduction is as follows.

Example 1.1. Consider the term rewriting rule $f(x) \to g(x,b)$, and let the term $t_1 = f(f(f(a)))$ be compressed as $C_1 \to f(\cdot)$, $C_2 \to C_1C_1$, $T \to C_2(T'), T' = f(a)$. A single term rewriting step on the compressed term t_1 by the rule $f(x) \to g(x,b)$ would produce $T' \to g(a,b)$, and hence the reduced and decompressed term is $f(f(g(a,b)))$. Other rewriting steps on the compressed term that do not decompress the term have to analyse the contexts. Let another term be $t_2 = f^{16}(a)$, compressed as $C_1 \to f(\cdot)$, $C_2 \to C_1C_1$, $C_3 \to C_2C_2$, $C_4 \to C_3C_3$, $C_5 \to C_4C_4$, $T \to C_5(a)$. A term rewriting step on T using $f(x) \to g(x,b)$ may rewrite the context $f(\cdot)$ and thus would produce $C_1 \to g(\cdot,b)$, and hence reduces the term in one blow to $g(\dots,(g(\dots,b)\dots),b)$. In fact, this is a form of a parallel rewriting step, see also Algorithm 6.1 (4).

The structure of the paper is as follows. First, the basic notions, in particular STGs, are introduced in Section 2. The first main part is an algorithm for linear submatching in Section 3. In Section 4, we analyse submatching for some special cases and also give a general algorithm for term submatching of compressed patterns and terms, and show that it can be performed in polynomial time for a fixed number of variables. Finally, in Section 6, we illustrate the application in term rewriting and equational deduction and show that n rewrites for a left-linear TRS can be performed in polynomial time.

2. Preliminaries

We will use standard notation for signatures, terms, positions and substitutions (see e.g. Baader and Nipkow 1998). Let $FVmult(s)$ be the set of variables occurring more than once in s. A position is a word over positive integers. For two positions p_1, p_2, we write $p_1 \leqslant p_2$, if p_1 is a prefix of p_2, and $p_1 < p_2$, if p_1 is a proper prefix of p_2. We call two strings w_1, w_2 *compatible*, if w_1 is a prefix of w_2, or w_2 is a prefix of w_1. We write $p[i]$ for the ith symbol of p, where 0 is the start index, and $p[i,j]$ for the substring of p starting at i ending at j. The set of free variables in a term t is denoted as $FV(t)$. Terms without occurrences of variables are called *ground*. A term where every variable occurs at most once is called *linear*. A *context* is a term with a single hole, denoted as $[\cdot]$. Sometimes, it is convenient to view a linear term containing one variable as a context, where the single variable represents the hole. As a generalization, a *multicontext* is a linear term,

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

where the variable occurrences are also called holes. Let $holep(c)$ be the position (as a string of numbers) of a hole in a context c, and let the *hole depth* be the length of $holep(c)$. If $c = c_1[c_2]$ for contexts c, c_1, c_2, then c_1 is a *prefix context* of c and c_2 is a *suffix context* of c. The notation $c[s]$ means the term constructed from the context c by replacing the hole with s. An n-fold iteration of a context c is denoted as c^n; for example, c^3 is $c[c[c]]$. A *substitution* σ is a mapping on variables, extended homomorphically to terms by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

Definition 2.1. A TRS R is a finite set of pairs $\{(l_i, r_i) \mid i = 1, \ldots, n\}$, called *rewrite rules*, usually written $\{l_i \to r_i\}$, where we assume that for all i : l_i is not a variable, and $FV(r_i) \subseteq FV(l_i)$.

A term rewriting step by R is $t \xrightarrow{R} t'$, if for some i: $t = c[\sigma(l_i)]$ and $t' = c[\sigma(r_i)]$ for some context c and some substitution σ.

This can also be seen as an equational deduction step, where the rules in R are the equational axioms.

2.1. *Tree grammars for compressions*

Definition 2.2. A *singleton context-free grammar (SCFG)* G, also called SLP is a 3-tuple $\langle \mathcal{N}, \Sigma, R \rangle$, where $\mathcal{N}$ is a finite set of non-terminals, Σ is a finite set of symbols (a signature) and R is a finite set of productions of the form $N \to \alpha$, where $N \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$. The sets $\mathcal{N}$ and Σ must be disjoint, each non-terminal X appears as a left-hand side of exactly one production of R, and $>_G$ on $\mathcal{N}$ is defined as $X >_G Y$ iff $X \to A \in R$ and Y occurs in A. The transitive closure $>_G^+$ must be irreflexive, i.e. there are no $>_G$-cycles. The word generated by a non-terminal N of G, denoted by $val_G(N)$ or $val(N)$ when G is clear from the context, is the word in Σ^* reached from N by successive applications of the productions of G. We omit a start symbol, but this is not essential.

An application for SLPs is the representation of compressed positions in compressed terms. We will use the well-known (polynomial-time) algorithms, constructions and their complexities on SLPs like equality check of compressed strings, computing prefixes, suffixes and the common prefix (suffix) of two strings asf (see Gasieniec *et al.* 1996a; Karpinski *et al.* 1995; Levy *et al.* 2008; Lifshits 2007; Plandowski 1994; Plandowski and Rytter 1999; Rytter 2004).

Definition 2.3. An *STG* is a 4-tuple $G = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$, where $\mathcal{TN}$ are tree/term non-terminals of arity 0, $\mathcal{CN}$ are context non-terminals of arity 1, and Σ is a signature of function symbols (the terminals), such that the sets $\mathcal{TN}$, $\mathcal{CN}$ and Σ are finite and pairwise disjoint. The set of non-terminals $\mathcal{N}$ is defined as $\mathcal{N} = \mathcal{TN} \cup \mathcal{CN}$. The productions in $\mathcal{R}$ must be of the following form:

— $A \to f(A_1, \ldots, A_m)$, where $A, A_i \in \mathcal{TN}$, and $f \in \Sigma$ is an m-ary terminal symbol.
— $A \to C_1 A_2$, where $A, A_2 \in \mathcal{TN}$, and $C_1 \in \mathcal{CN}$.
— $C \to [\cdot]$, where $C \in \mathcal{CN}$.
— $C \to C_1 C_2$, where $C, C_1, C_2 \in \mathcal{CN}$.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

— $C \to f(A_1, \dots, A_{i-1}, [\cdot], A_{i+1}, \dots, A_m)$, where $A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_m \in \mathcal{TN}$, $C \in \mathcal{CN}$, and $f \in \Sigma$ is an m-ary terminal symbol.
— $A \to A_1$ (λ-production), where A and A_1 are term non-terminals.

Let $N_1 >_G N_2$ for two non-terminals N_1, N_2, iff $(N_1 \to t) \in \mathcal{R}$, and N_2 occurs in t. The STG must be non-cyclic, i.e. the transitive closure $>_G^+$ must be irreflexive. Furthermore, for every non-terminal N of G there is exactly one production having N as left-hand side. Given a term t with occurrences of non-terminals, the derivation of t by G is an exhaustive iterated replacement of the non-terminals by the corresponding right-hand sides. The result is denoted as $val_G(t)$. We will write $val(t)$ when G is clear from the context. In the case of a non-terminal N of G, we also say that N (or G) *generates* $val_G(N)$ or *compresses* $val_G(N)$. The depth of a non-terminal N is the maximal number of $>_G$-steps starting from N, and the depth of G is the maximal depth of all its non-terminals. The size of an STG is the number of its productions, denoted as $|G|$.

Definition 2.4. Let G be an STG and V be a set of variables. Then, (G, V) is an *STG with variables*, where additional production forms are permitted:

— $A \to x$, where $A \in \mathcal{TN}$ and $x \in V$.
— $x \to A$ (λ-production), where $x \in V$ and $A \in \mathcal{TN}$.

This means that variables may be terminals or non-terminals, depending on the existing productions. The measure $Vdepth(N, V)$ is defined as the maximal number of $>_G$-steps starting from N until an element of V or a terminal is reached.

In the following, we always mean STG with variables if variables are present.

Definition 2.5. An STG G is called a DAG, if there are no context non-terminals. The terms $val(S)$ for term non-terminals S of G are also called *DAG-compressed*.

Note that the term depth of DAG-compressed terms is at most the size of the DAG, whereas the term depth of STG-compressed terms may be exponential in the size of the STG. Note also that every subterm in a DAG-compressed term is represented by a non-terminal, whereas in STG-compressed terms, there may be subterms that are only implicitly represented.

2.2. *Grammar extensions*

We list the main grammar extensions required in this paper and give also their estimations for the size increase.

Definition 2.6 (Grammar extension). We say that the STG with variables (G', V') is an extension of the STG with variables (G, V), where $G = (\mathcal{T}, \mathcal{C}, \Sigma, \mathcal{R})$, $G' = (\mathcal{T}', \mathcal{C}', \Sigma, \mathcal{R}')$ denoted $(G', V') \supseteq (G, V)$, if $\mathcal{R} \subseteq \mathcal{R}'$, $\mathcal{C} \subseteq \mathcal{C}'$ and $V \subseteq V'$.

We repeat the constructions and their properties (see Gascón *et al.* (2011); Levy *et al.* (2011)).

Lemma 2.7. Let G be an STG with variables.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Term-construction. Let $f \in \Sigma$ be an n-ary function symbol and assume there are n terms $t_1, \ldots, t_n$ that are compressed by G. Then, there exists a grammar extension $G' \supseteq G$ generating the context $f(t_1, \ldots, [\cdot], \ldots, t_{n-1})$ and also a grammar extension generating the term $f(t_1, \ldots, t_n)$ satisfying

$$|G'| \leqslant |G| + 1,$$
$$Vdepth(G', V) \leqslant Vdepth(G, V) + 1.$$

Concatenation. Let the contexts $c_1, \ldots, c_n$ for $n \geqslant 1$ be generated by G. Then, there exists a grammar extension $G' \supseteq G$ that generates the context $c_1[c_2[\ldots[c_n]\ldots]]$ and satisfies

$$|G'| \leqslant |G| + n - 1,$$
$$Vdepth(G', V) \leqslant Vdepth(G, V) + \log n + 1.$$

Exponentiation. Let the context c be generated by G. For any $n \geqslant 1$, there exists a grammar extension $G' \supseteq G$ that generates the context c^n and satisfies

$$|G'| \leqslant |G| + 2\log n,$$
$$Vdepth(G', V) \leqslant Vdepth(G, V) + \log n + 1.$$

Prefix and suffix. Let the context c be generated by G. For any non-trivial prefix or suffix c' of the context c, there exists a grammar extension $G' \supseteq G$ that generates c', and satisfies

$$|G'| \leqslant |G| + depth(G) - 1,$$
$$Vdepth(G', V) \leqslant Vdepth(G, V) + \log(depth(G)) + 1.$$

Subterm. Let the context c or term t be generated by G. For any non-trivial subterm t' of the context c or of the term t, there exists a grammar extension $G' \supseteq G$ that generates t' and satisfies

$$|G'| \leqslant |G| + depth(G),$$
$$Vdepth(G', V) \leqslant Vdepth(G, V) + \log(depth(G)) + 2.$$

Subcontext. Let the term t be generated by G. For any non-trivial prefix context c of the term t, there exists a grammar extension $G' \supseteq G$ that generates c and satisfies

$$\begin{aligned} |G'| &\leqslant |G| + depth(G)(depth(G) + 3/2), \\ Vdepth(G') &\leqslant Vdepth(G) + 2\log(depth(G)) + 4. \end{aligned}$$

Instantiation. Let the term t be generated by G, and let $x \in V$ be a terminal and let N be a term non-terminal. Then, the grammar extension $G' \supseteq G$ with the additional production $x \mapsto N$ satisfies

$$\begin{aligned} |G'| &\leqslant |G| + 1, \\ Vdepth(G') &= Vdepth(G). \end{aligned}$$

Lemma 2.8. For an STG with variables, we have $depth(G) \leqslant (Vdepth(G, V) + 1) \cdot (|V| + 1)$.

Definition 2.9 (Grammar extension step). We say that the pair (G', V') is constructed from the pair (G, V) using an α-*bounded grammar extension step* if it can be constructed by term-construction, concatenation, exponentiation, prefix, suffix, subterm, subcontext

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

or instantiation, where the exponent used for exponentiation is bounded by 2^α, and the number of concatenated contexts is bounded by α.

Lemma 2.10. If G' is an α-bounded extension of G according to Definition 2.9, then the following inequations hold:

$$\begin{aligned}|G'| &\leqslant |G| + \mathcal{O}(depth(G)^2) + \mathcal{O}(\alpha),\\ Vdepth(G') &\leqslant Vdepth(G) + \mathcal{O}(\log(depth(G))) + \mathcal{O}(\log \alpha).\end{aligned}$$

In Levy *et al.* (2011), it is shown that a polynomial number of grammar extension as above under certain restrictions for concatenation and exponentiation leads to a polynomial-sized grammar.

Theorem 2.11. If the grammar G has size $|G| = \mathcal{O}(n)$, and (G', V') is constructed from $(G, \emptyset)$ using $\mathcal{O}(n^k)$ many $\mathcal{O}(n)$-bounded grammar extension steps, then

$$\begin{aligned}&|G'| = \mathcal{O}(n^{5k+2}),\\ &depth(G') = \mathcal{O}(n^{2k+1}),\\ &Vdepth(G', V') = \mathcal{O}(n^{k+1}),\\ &|V'| = \mathcal{O}(n^k).\end{aligned}$$

The extension steps above can be performed in polynomial time, where proofs can be found in Gascón *et al.* (2011), and missing ones can be easily derived from these proofs.

Proposition 2.12. Let G be an STG and S be a term non-terminal. Then, G can be transformed in linear time into an STG G', such that $val_G(S) = val_{G'}(S)$, where every production for term non-terminals in G' is of the form $A \to a$ for a terminal a, or $A \to CA'$ and $A' \to a$ for a terminal a.

Proof. The modification of the grammar can be done bottom-up in the grammar: If there is a production $A \to CA'$, then we can assume that $A' \to a$, or $A' \to C'A''$, where $A'' \to a'$ for a terminal. Then, we replace the production for A by $A \to C''A'', C'' \to CC'$. □

2.3. *Submatching*

Given two first-order terms s, t, where s (the pattern) may contain variables, the submatching problem is to identify an instance of s as a subterm of t. The submatching (also called encompassment relation) is a prerequisite for term rewriting.

Definition 2.13. The *compressed term submatching problem* is as follows:

Let s be a term that may contain variables, and let t be a (ground) term, where both are compressed with an STG $G = G_S \cup G_T$, such that $val(T) = t$ and $val(S) = s$ for term non-terminals $S \in G_S$, $T \in G_T$. Sometimes, we assume that G_S, G_T are disjoint. The task is to compute a (compressed) substitution σ such that $\sigma(s)$ is a subterm of t; also, the (compressed) position (all positions) p of the match in t should be computed. Specializations of the submatching problem are as follows:

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

uncompressed: If s is given as a plain term without any compression.
ground: If s is ground.
DAG-compressed: If s is DAG-compressed.
linear: If s is a linear term, i.e. every variable occurs at most once in s.

Note that (compressed) term matching is a special case: It asks whether there exists a substitution such that $\sigma(s) = t$, which can be answered in polynomial time (Gascón *et al.* 2008, 2011).

We derive the following lemma on the possible submatching positions by an easy case analysis.

Lemma 2.14. Let G be an STG, s be a term and let T be a non-terminal with $val_G(T) = t$, where t is ground. If there is some substitution σ, such that $\sigma(s)$ is a subterm of t, then there exist following possibilities:

1 There is a term non-terminal B of G such that $val_G(B) = \sigma(s)$.
2 There is a production $B \to CB'$ in G, such that $\sigma(s) = c[val_G(B')]$, where c is a non-trivial suffix context of $val_G(C)$. There are subcases for the hole position p of c.
   (a) (overlap case) p is a position in s.
   (b) $p = p_1 p_2$, where p_1 is the maximal prefix of p that is also a position in s. Then, $s_{|p_1} = x$ is a variable. The algorithms below have to distinguish further subcases:
      i (subterm case) x occurs more than once in s,
      ii (subcontext case) x occurs exactly once in s.

Example 2.15. The number of possible substitutions for a submatch may be exponential: Let the productions be $S \to f(x)$, and $T \to C_n[a], C_0 \to f([.]), C_1 \to C_0C_0, \ldots C_i \to C_{i-1}C_{i-1}$. Then, $val(T) = f^{2^n}(a)$, and every substitution $\sigma(x) = f^i(a)$ with $0 \leqslant i \leqslant 2^n - 1$ corresponds to a submatch. However, distinguishing the subterm (Lemma 2.14, case 2a) and subcontext case (Lemma 2.14, case 2b), we see that the exponentially many substitutions correspond to a single output for the subcontext case.

3. Term submatching with linear terms

In this section, we describe a polynomial algorithm for compressed term submatching, where the pattern term is linear.

3.1. *Overlaps of strings with character-holes*

We will need results for so-called partial words and their overlaps as a prerequisite for results on a multiple overlap of a linear term (multicontext) with itself.

A partial word w is a word over $\Sigma \cup V$, where Σ is an alphabet and V is a set of variables, and where every variable occurs at most once in w. The variables are also considered as character-holes (notation: $\circ$), i.e. the variables are substitutable only with single characters. We analyse overlapping partial words and will apply the obtained results to overlaps of a linear term with itself.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

We repeat the definition and the theorem from Schmidt-Schauß (2012).

If for a partial word s there is a number $p < |s|$ such that $s[i] = s[i+p]$ for all i, where $s[i], s[i+p]$ are defined and are not holes, then we say that s is *locally periodic* with period p. For example, the partial word $ababa\circ acac$ is 2-locally periodic. If there is a number $p < |s|$ such that $i \equiv j \bmod p$ implies $s[i] = s[j]$ for all $0 \leqslant i, j \leqslant |s| - 1$ if $s[i], s[j]$ are defined and not holes, then s is called *periodic* (also strongly periodic), and p is a *period* of s.

Definition 3.1 (Schmidt-Schauß 2012). Let s be a partial word and $n \geqslant 2$. An n-fold *overlap* of s (with itself) is given by starting positions $0 \leqslant p_1 < p_2 < p_3 < \cdots < p_n \leqslant |s| - 1$, such that for all $i, j = 1, \ldots, n$ and $0 \leqslant k \leqslant |s| - 1$: if $0 \leqslant k - p_i, 0 \leqslant k - p_j$, then $s[k - p_i] = s[k - p_j]$, provided neither $s[k - p_i]$ nor $s[k - p_j]$ is a hole.

If not stated otherwise, we assume that $p_1 = 0$.

Theorem 3.2 (Schmidt-Schauß 2012). Let w be a partial word with n holes, and assume that there is an overlap of $m \geqslant n+2$ occurrences of w. Let p_{max} be $\max\{p_{i+1} - p_i \mid i = 1, \ldots, m-1\}$. Assume $|w| - p_m \geqslant 2n \cdot p_{max}$; this means there are $2n \cdot p_{max}$ common positions of all occurrences of w.

Then, the partial word w is periodic, and a period is $p_{all} := gcd(p_2 - p_1, p_3 - p_2, \ldots, p_m - p_{m-1})$. Moreover, the overlap is consistent with using the same substitution for every occurrence of w.

3.2. *Overlaps of multicontexts*

For a linear term (a multicontext) s and two positions p_1, p_2 of s with $p_1 < p_2$, we write $s[p_1, p_2]$ for the (multi-)context starting at p_1 with hole at p_2, i.e. $s[p_1, p_2] := s[[\cdot]/p_2]_{|p_1}$, i.e. for the context constructed from s by first replacing the subterm at p_2 by a hole, and then selecting the subcontext at p_1.

Let c be a multi-context, let p be a path to a hole of c and let there be a number $m < |p|$ such that for all $0 \leqslant i, j \leqslant |p| - 1$: $i \equiv j \bmod m$ implies that $c[p[0, i], p[0, i+1]]$ and $c[p[0, j], p[0, j+1]]$ are unifiable (as linear terms); then, c is called *periodic* (also strongly periodic) (along p) and m is a *period* of c (along p). If additionally, there is a position q such that p is a prefix of q^k for some positive integer k, then we say that c is periodic along q^∞.

From Theorem 3.2, we derive a theorem for periodicity of multicontexts if there is a sufficiently dense overlap. Note that for the overlap, we first have to select a direction.

Definition 3.3. Let c be a multi-context with $h \geqslant 1$ holes. Let p be the position of a fixed hole of c, and let $p_i, i = 1, \ldots, n$ be prefixes of p such that $i < j$ implies $p_i < p_j$ with $n \geqslant h+2$. Assume that there are n copies of c starting at position p_i such that p is a prefix of $p_i p$, i.e. the hole position of c starting at p_i is compatible with p for all i.

— This is an *overlap with a cut* if for all positions that are also positions in c at p_0, equality must hold, i.e. parts of other c that are positioned in the hole of the first c are ignored. This corresponds to the overlap definition in Schmidt-Schauß (2012) and Definition 3.2.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

— This is a *full overlap* if for all common positions, equality must hold, i.e. also subterms in the hole of the first c have to be compared. Note that this is also an overlap with cut.

Theorem 3.4 (Periodicity theorem). Let c be a multi-context with $h \geqslant 1$ holes. Let there be an overlap with cut of n copies of c with positions p, p_i as in Definition 3.3. Let p_{max} be $\max\{|p_{i+1}| - |p_i| \mid i = 1, \ldots, n-1\}$, and assume $|p| - |p_n| \geqslant 2h \cdot p_{max}$; which means there are $2h \cdot p_{max}$ common positions on the path p of all occurrences of c.

Then, the multicontext c is periodic (in the direction p), and a period length is $p_{all} := gcd(|p_2| - |p_1|, |p_3| - |p_2|, \ldots, |p_n| - |p_{n-1}|)$. Moreover, the overlap is consistent with using the same substitution for the variables for every occurrence of c.

If it is a full overlap, then for the indices i with $h + 2 \leqslant i \leqslant n - h + 2$, the substitutions into the variables of c are identical, with the possible exception of the hole-variable at p.

Proof. We translate c into a partial word w as follows: Along the path p, we split c into the contexts $c_1, \ldots, c_m$ with $m = |p|$ and such that c_i are multi-contexts with a hole at depth 1 and such that the hole path of this hole of c_i is exactly the integer in p at position i. Then, $c = c_1 \ldots c_m$. Now, there are at most $h - 1$ contexts among the c_i that have more than one hole. The partial word w is constructed as $w = w_1 \ldots w_m$, where $w_i := c_i$ if c_i is a context with exactly one hole, and $w_i := X_i$ if c_i is a context with at least two holes. The variables X_i stand for contexts with the same hole path as c_i. Now, w can be viewed as a partial word where the variables X_i act as character-variables and so we can apply results from partial words. Now, we have a multiple overlap of the partial word w with itself and can apply Theorem 3.2, which shows periodicity of w. The periodicity of w and the consistent instantiation into the holes of w then imply a periodicity of c, since we have started with an overlap. The overlap enforces that every variable X_i of w is instantiated with a context $c_{i'}$ that is ground. Since we have assumed that we have a multiple overlap, and since every hole must be instantiated, the condition on unifiability in the periodicity definition is satisfied, also for the multicontexts c_i with more than one hole.

The condition for full overlaps is satisfied, since every instantiation in the range is forced by one subterm in the period subcontext. □

3.3. *Overlaps of linear terms and contexts*

We consider the overlap of multicontexts $c, c_1, c_2, \ldots$ and a context d in this section. In particular special variants of overlaps have to be analysed: Overlaps where the hole of d is not compatible with any hole of c. The overlaps where a hole of c is compatible with a hole of d are later dealt with in a standard fashion.

This exhibits a difference between strings and linear terms: Periodicities in linear terms are not only possible along a hole-path but also along other paths, and there are two different kinds of such periodicities (see Proposition 3.7), which for contexts (linear terms with a single variable) already appeared in Schmidt-Schauß (2005).

Definition 3.5. Let c be a multicontext with at least one hole, and let d be a context with exactly one hole. An *overlap* of c with d is an occurrence of c starting at a position

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

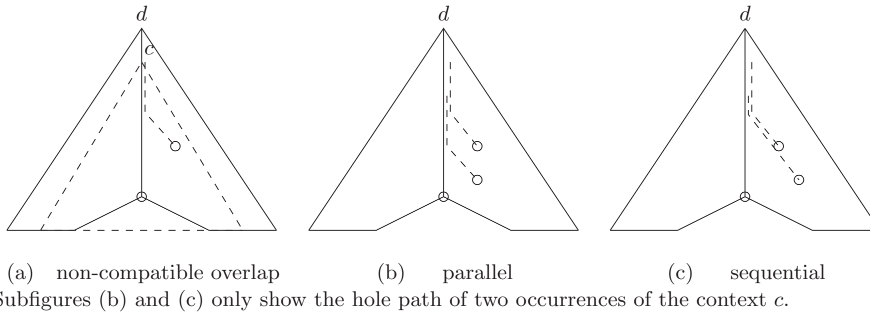

Fig. 1. Non-compatible, parallel and sequential overlap of c with d. (a) Non-compatible overlap. (b) Parallel. (c) Sequential. Subfigures (b) and (c) only show the hole path of two occurrences of the context c.

p of d, where p is a prefix of the hole-path $holep(d)$. More exactly, there is a ground substitution σ, a ground term r and a ground context d_0 that is a prefix context of d, such that $d_0[\sigma(c)] = d[r]$. A *compatible overlap* of c with d is an occurrence of c with starting position p, where p is a prefix of the hole-path $holep(d)$, and for some hole position q of c, pq is compatible with $holep(d)$. If for every hole position q of c, pq is not compatible with $holep(d)$, then the overlap is called *non-compatible*. For a non-compatible overlap, let the *maximal common hole path (mchp)* be the maximal path q, such that pq is a prefix of $holep(d)$, and q is a prefix of some hole path of c.

Example 3.6. Let $d = f(a_1, f([\cdot], a_1))$ and let $c = f(a_1, [\cdot])$. Then, c overlaps d at position ε, which is a compatible overlap, since the start as well as the hole position of c is on the hole path of d. The overlap of c with d at position 2 (in d) is a non-compatible overlap, since the hole of c is at 2.2, which is not a prefix or suffix of the hole path of d, which is 2.1. Further examples for non-compatible overlaps can be found in Example 3.8.

Note that overlaps of c with d at other positions than on the hole path occur in the bookkeeping of overlaps, but are tabled as overlaps with other contexts (i.e. subcontexts) or terms.

Proposition 3.7. Let c be a multicontext with at least one hole, and let d be a context with exactly one hole, and let $p_1 < p_2$ be two positions of non-compatible overlaps of c in d. Let q_i be the mchp of c at p_i for $i = 1, 2$.

Then, there are the following two cases (see Figure 1):

1 $q_1 = q_2$ (the *parallel overlap* case). Then, for p' such that $p_1 p' = p_2$ the path $p_1(p')^n$ is compatible with $holep(d)$ for all n. Also, this is a multiple overlap of c' with itself at positions $(p')^i$, where c' is constructed from c with an extra hole at p'', where $p_1 p'' = holep(d)$.

2 $q_2 < q_1$ (the *sequential overlap* case). Then, $p_2 q_2 = p_1 q_1$, i.e. there is a fixed position on the hole path of d, where the hole paths of occurrences of c deviate.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Proof.

1 Let $q_1 \leqslant q_2$, and assume for contradiction that $q_1 < q_2$. For simplicity, we assume that $q_i = \varepsilon$, the case for non-empty q_i is similar. Then, $c = f(\dots, c_k, \dots,)$ and p_1k is a prefix of $holep(d)$, where c_k does not contain holes of c. The term at position p_1p' in d is also equal to c_k, since we have assumed $q_1 < q_2$; hence, we have a contradiction. Thus, $q_1 = q_2$, and the term at position p_1p' in d is also equal to c_k up to the hole of d. Thus, $p_1(p')^n$ must hit the hole of d, since otherwise there is an infinite path in c.

2 Now, let $q_2 < q_1$. Again we assume for simplicity that $q_2 = \varepsilon$. Note that p_2q_2 and p_1q_1 are compatible. We exclude $p_2q_2 \neq p_1q_1$ by showing that the following cases are impossible:

— $p_1q_1 < p_2$ is not possible: Assume this holds, then the term $t = c_{|q_1}$ does not contain a hole. Moreover, it is contained in a term c_i, where $c = f(c_1, \dots, c_n)$ and p_2i is not a prefix of $holep(d)$. Looking at the second overlap of c, we see that t is properly contained in itself.

— $p_2 < p_1q_1$ is impossible: We look for a construction of positions that leads to an infinite path. Let p' such that $p_1p' = p_2$. Then, we start the walk at p_1 with p'. It arrives at p_2. Now, we again look into the occurrence of c that has its top at p_2. Again we can walk p' and now reach a position in c that is not a prefix of $holep(d)$. The reason is that c at p_2 has its holes not above the hole of d. But then, we can repeat this walking infinitely, which is a contradiction.

□

Example 3.8. Let $c' = f(f(a_1, a_2), [\cdot])$ be a context, $c = f(f(x, y), (c')^{100}[.])$, and let $d = (c')^{100}[\cdot]$. Then, there is an overlap of c with d at positions $\varepsilon, 2, 2.2, \dots$ It is an overlap of the first kind, i.e. a parallel overlap.

An overlap of the second kind, i.e. a sequential overlap is the following: Let $c = f(a_1, f(a_1, f(a_1, [\cdot])))$ and let $d = f(a_1, f(a_1, f(a_1, f([\cdot], f(a_1, f(a_1, a_1))))))$. Then, the overlap positions are $\varepsilon, 2, 2.2, 2.2.2$.

Remark 3.9. If there are three or more non-compatible overlaps of a multicontext c with a context d, then there are only two possible configurations:

— a parallel overlap,
— a sequential overlap.

Mixtures of them are not possible for the non-compatible overlaps. The argument is as follows: look for the shortest mchp q_i. If it is the first one, then it must be a parallel overlap for all occurrences. If the shortest mchp q_j is not the first one, then the one with $k > j$ are parallel w.r.t. overlap j, but then the overlap for $i = 1$ has to be sequential w.r.t. j and $j + 1$, which is impossible.

3.4. *Tabling prefixes of multicontexts in contexts*

In this section, we define the algorithm for constructing tables in a dynamic programming style, where the tables represent the prefix matchings of a multicontext c in a context d.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The construction is similar to the string matching with character-variables in Schmidt-Schauß (2012). Note that matchings of c in $val(A)$ for non-terminals A given a compressed position can be recognized in polynomial time using the compressed matching algorithm in Gascón *et al.* (2011).

Let G be an STG, and let S be a non-terminal in G representing the linear term $s = val(S)$. Let G_t be an STG, T be a non-terminal in G_t, with $t = val(T)$, which is the term in which we are searching for a match of s. In order to find the matches, a complete table is computed for all prefix matches of s in all non-terminals T' that occur in a derivation of T.

For convenience, we will assume that in the used STGs, the right-hand sides CA of productions of term non-terminals only permit $val(A) = a$, where a is a constant. This is possible according to Proposition 2.12 without much overhead.

In the following, we say that A *compatibly matches* B for a term non-terminal A and a context non-terminal B, if there is a substitution σ, such that $\sigma(val(A)) = val(B)[r]$ for some term r. If there is a substitution σ, such that the match is exact, i.e. $\sigma(val(A)) = val(B)$, then we say that A *exactly matches* B. Here, we allow that the substitution may introduce a single hole into $\sigma(val(A))$.

Definition 3.10. We define a *result table* and $h + 1$ *prefix tables*, with coordinate A for context non-terminals A of G_t, where h is the number of variables (number of holes) in $s = val(S)$, and the extra table is for non-compatible overlaps. For every $0 \leqslant i \leqslant h$, there is a table of prefixes where the ith hole of $val(S)$ and the hole of $val(A)$ are compatible, and there is a table for the prefixes with non-compatible overlap. We explain the following two kinds of tables:

— A *result table* that contains entries C or (C, P, n), where C and P are context non-terminals for ground contexts, n is a number, and the hole path of $val(C)$ is a prefix of the hole path of $val(A)$, such that

- for the entry C, CS matches A exactly;
- for the entry (C, P, n), for $k = 0, \ldots, n$, every CP^kS matches A exactly.

— A *prefix table* that contains the following two different kinds of entries:

- C where the hole path of C is a prefix of the hole path of $val(A)$, such that CS compatibly matches A;
- (C, P, n), (we allow ∞ for n), where the hole path of $val(CP^i)$ is compatible with the hole path of $val(A)$ for all $i = 0, \ldots, n$. This means that CP^i compatibly matches A for all i.

  The third component n is only a number for the table of the non-compatible overlaps that are sequential overlaps, and otherwise it is ∞.

  The generation of the entry will guarantee the following periodicity claims, since only compaction can generate such entries, and this in turn is only possible if the periodicity theorem is applicable:

  1 If it is an entry in a table corresponding to a hole of S, then the context c' with $val(A) = val(C)[c']$ is periodic in the direction $(holep(val(P)))^{\infty}$, and also $val(S)$ is periodic in the direction $(holep(val(P)))^{\infty}$.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

2 If it is an entry in a non-compatible table corresponding to a hole of S, then the context c' with $val(A) = val(C)[c']$ is periodic in the direction $(holep(val(P)))^\infty$, and also $val(S)$ is periodic in the direction $(holep(val(P)))^\infty$.
If the third component is n, then s' is periodic, which is $val(S)$ cut at the hole position of $holep(val(P))^n$, i.e. $s' := val(S)[[\cdot]/p]$, where $p = (holep(val(P)))^n$.
If the third component is ∞, then the context c' with $val(A) = val(C)[c']$ is periodic in the direction $(holep(val(P)))^\infty$, and also $val(S)$ is periodic in the direction $(holep(val(P)))^\infty$.

Example 3.11. We describe several small examples for compatible entries in a prefix table. Therefore, we slightly extend Example 3.8. Let the STG be $S \to A_2; A \to A_1A_1; A_1 \to A_2A_2, A_2 \to f(a_1, [\cdot])$.

1 Then, (C, A_2, ∞) for $C \to [\cdot]$ is a potential entry in a result table for A.
2 Let $A_4 \to g([\cdot])), B \to A_4A, C' \to A_4$. Then, (C', A_2, ∞) is an entry in the result table for B.
3 Let $B' \to BA_4$, then $(A_4, A_2, 2)$ is an entry in the result table for B'.
4 The tuple $(A_4, A_2, 3)$ is an entry in the prefix table for B.
5 Let $B'' \to A_6A_4, A_6 \to A_4A_1$. The context A_6 is then a potential entry in the result an prefix tables of B''.

Note that item 4 cannot be used as a result, since composing B as in $B' \to BA_4$ in item 3 may render an overlap invalid.

Example 3.12. We describe an example for a non-compatible entry in a prefix table. Therefore, we slightly modify Example 3.8. Assume there is an STG G. Let $c = f(a_1, f(a_1, f(a_1, f(a_1, [\cdot]))))$, $d = f(a_1, f(a_1, f(a_1, f([\cdot], f(a_1, f(a_1, a_1))))))$, and let P, D, C_0, S be a non-terminals such that $val(P) = f(a_1, [\cdot])$, $val(D) = d, val(S) = c$, $val(C_0) = [\cdot]$. Then, an entry in the non-compatible prefix table for D could be $(C_0, P, 3)$.

In the construction of the tables, we detail the different treatments of the hole-i table and the non-compatible table if necessary. Note that during construction of the tables, the STG G may be extended to G'. Note that the single entries C are explicitly constructed, whereas the entries (C, P, n) indicating periodicity are only generated in the compaction step. The construction of the prefix table in the case $A \to A_1A_2$ and the periodic cases is depicted in Figure 3 where (a) shows the case where A has a periodic suffix, (b) shows the case where A has an inner part that is periodic, (c) shows a case where the periodicity goes into a direction that is not compatible with the hole of A_2, which leads to the sequential overlap case and (d) is a case of a sequential overlap already in the table for A_1.

Note that we do not give every detail: For example, length computations in STGs can be done in linear time; positions are always SLP-compressed. For compressed strings, it is known how to compare them for equality, and how to compute common prefixes of two compressed strings, in polynomial time. Similar for contexts, operations of the following kind are performed several times: Given a non-terminal S such that $s = val(S)$ is a linear term and P is a ground context, find a maximal k, such that $val(P)^k$ is a prefix of s (or such that s compatibly matches $val(P)^k$, respectively). This is done as follows: First, compute the size of s, which gives an upper bound for k. Then, construct a non-terminal

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

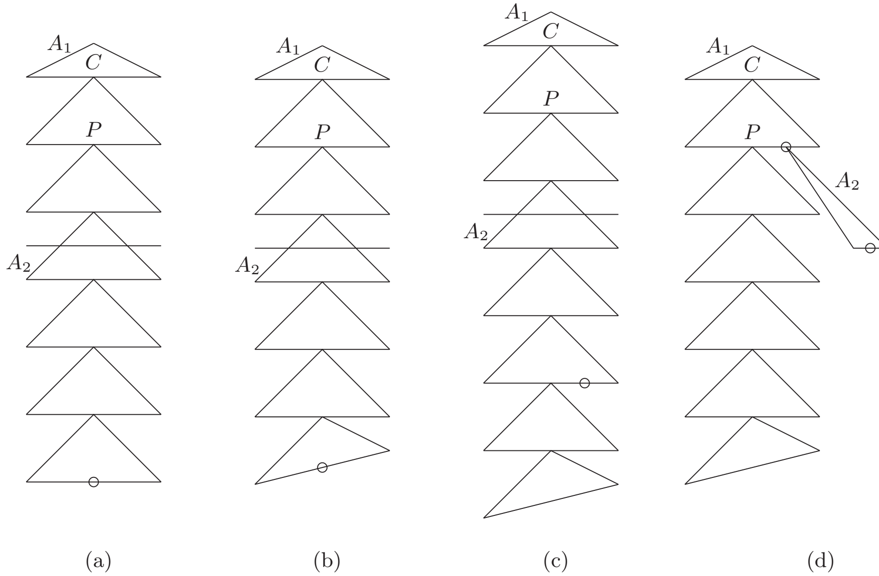

Fig. 2. Cases in the construction of the prefix tables for periodic entries.

P_k with $val(P_k) = val(P)^k$, and apply a prefix test to S and P_k. Use a interval bisection for the interval $\{1,\ldots,k\}$ to iterate the test until the maximum is found. These algorithms can be performed in polynomial time.

Algorithm 3.13 (Construction of suffix and prefix tables for overlaps of linear terms with context-non-terminals). We construct the table entries for all context non-terminals A in G, and where S is the fixed term non-terminal for the linear term $s = val(S)$. The construction is bottom-up w.r.t. the STG. If the tables for a context non-terminal A are constructed with grammar production $A \to A_1A_2$, then the tables are already filled for A_1 and A_2, and for every entry, one of the five cases may apply. The final step for A is then the compaction step below for every prefix table of A.

The *first case* is that the hole depth of $val(A)$ is 1.

If S does not compatibly match A, then there are no entries in the tables. If S exactly matches A, then the empty context is in the result table. If S does not exactly match A, but compatibly matches A, then the empty context is in the prefix tables: If $holep(val(A)) = k$, and the ith hole-path of $val(S)$ starts with k, then the entry is in the prefix table i. If there is no such hole-path of $val(S)$, then the entry is in the non-compatible prefix table.

The *second case* is that $A \to A_1A_2$ is the production for A, inheriting entries of A_2.

For every entry C or (C, P, n) in the prefix table of A_2, there is an entry C' or (C', P, n), respectively, in the prefix table with new production $C' \to A_1C$.

The *third case* is that $A \to A_1A_2$ is the production for A, inheriting an entry C in the prefix table of A_1.

If CS exactly matches A, then there is an entry C in the result table of A.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Otherwise, if CS compatibly matches A, then we insert C into some prefix tables of A. The exact tables that are to be filled are determined according to the relative positions of the hole of $val(CS)$ and $val(A)$.

The *fourth case* is that $A \to A_1A_2$ is the production for A, and that (C, P, n) is an entry in the prefix table of A_1 where n is an integer. This is case (d) of Figure 3.

Since n is a number, it is an entry in the non-compatible overlap table, for a sequential overlap. In this case, we only have to check whether the leftmost single entry represented by the sequence can be extended to A_1A_2: That is, only the check whether CS compatibly matches A_1A_2 has to be performed. If yes, then the entry is also in the (same) prefix table for A.

The *fifth case* is that $A \to A_1A_2$ is the production for A, and that (C, P, n) is an entry in the prefix table of A_1, where n is ∞.

Note that for all these cases, only start positions of S need to be considered, which are in A_1, since the other ones are already inherited.

Then, we first construct context non-terminals P_1, P_2, P' with $val(P) = val(P_1P_2)$, such that there is a k with $val(CP^kP_1) = val(A_1)$ and with $P' \to P_2P_1$. Figure 3 indicates the cases for the periodicities in $val(A)$. The case distinction also has to take care of the periodicities (as a multicontext) of $val(S)$ and of prefixes of $val(S)$. There are several cases, listed as follows:

1 $val(A_2)$ is a prefix of $val(P')^i$ for some i. This is case (a) of Figure 3.
Then, a (maximal) context non-terminal P_3 and the maximal m can be computed (using interval bisection and matching) such that $val(A_2) = val((P')^mP_3)$ and $val(P_3)$ is a prefix of $val(P')$. Also, the maximal context non-terminal P_4 is computed by interval bisection, such that $val(P_4)$ is a prefix of $val(P)$ and $val(CP^{k'}P_4) = val(A)$. The cases are as follows:

(a) $val(S)$ is periodic in the direction $val(P)^\infty$: There is some h such that S exactly matches P^hP_4.
Compute the minimal h such that S exactly matches P^hP_4. This computation is possible by interval bisection, since $val(A_2) = val((P')^mP_3)$, and since $val(S)$ is periodic in the direction $val(P)^\infty$. Then, the entry in the result table is $(C, P, k' - h)$. If $h \neq 0$, then for the exponents $h' < h$, S must compatibly match; hence, we add the entry (C', P) to the prefix table, where $C' \to CP^{h+1}$ is the production for C'.

(b) Item 1a does not hold, but there is some h such that S compatibly matches P^hP_4. Then, compute the maximal h such that S compatibly matches P^hP_4. This computation is possible by interval bisection. The entry in the prefix table is (C', P) with $C' \to CP^{k'-h}$, which is an entry for a non-compatible (parallel) overlap. There is no entry in the result table.

2 $val(A_2)$ is not a prefix of $val(P')^i$ for any i. $(P')^{k_1}$ matches A_2 exactly, where k_1 is chosen maximal. The condition is that $(P')^{k_1+1}$ does not match A_2 compatibly. This is case (b) of Figure 3.
Note that there is some redundancy compared with the next item.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Then, compute a context non-terminal P_3 such that $val((P')^{k_1}P_3) = val(A_2)$. There are $k + k_1 + 1$ full periods of P in $val(A)$ on the hole path starting from the hole of C. There are several cases.

(a) $val(S)$ is periodic in the direction $val(P)^\infty$, i.e. S can be split into $S = S'_0S_1S'_1S_2S'_2 \dots S_mS'_m$, where S_i, S'_i are fresh context non-terminals, $val(S_i) = val(P)^{j_i}$ for some $j_i \geqslant 0$, and $holep(val(S'_i)) = holep(P)$ for all i, S'_i matches P for all i, and $m + 1 \leqslant |FV(val(S))| - 1$.
If $h \leqslant k + k_1 + 1$, then S exactly matches D if $val(CP^iD) = val(A_1A_2)$ for all appropriate i, and hence there is a new entry in the result table: $(C, P, k + k_1 - h)$. For every entry $CP^{k'}$, such that the position of S'_i is exactly at the start position of P_3, an extra match test has to be made, whether or not it is in the result table, or in the prefix table, and if yes in which prefix table.
Note the number of these tests is at most the number of holes in $val(S)$.

(b) Only a prefix of $val(S)$ is periodic in the direction $val(P)^\infty$, i.e. S can be split into $S = S'_0S_1S'_1S_2S'_2 \dots S_mS'_mR$, where S_i, S'_i, R are fresh context non-terminals, $val(S_i) = val(P)^{j_i}$ for some $j_i \geqslant 0$, and $holep(val(S'_i)) = holep(P)$ for all i, S'_i matches P for all i, and $m+1 \leqslant |FV(val(S))|-1$, and m is maximal. Then, there is no periodic entry. For every potential entry $CP^{k'}$, such that the position of S'_i or R is exactly at the start position of P_3 an extra match test has to be made, whether or not it is in the result table, or in the prefix table, and if yes in which prefix table.
Note the number of these tests is at most the number of holes in $val(S)$.

3 Items 1 and 2 do not hold. Then, $val(A_2) = val(P_2)val(P)^{k_1}d_5val(P)^{k_2}P_6$, where d_5 is a multicontext with two holes, one of its holes has position $holep(val(P))$ and the other hole corresponds to the hole of $val(A_2)$, $val(P)$ matches d_5, and P_6 is a term non-terminal without prefix $val(P)$. This is case (c) of Figure 3.
There are $k+1+k_1+1+k_2$ full periods of P in $val(A)$ on the hole path starting from the hole of C, with one exception at the hole of $val(A_2)$: The contribution of $val(P)$ may only fit after an instantiation into the hole of $val(A_2)$. The 2-hole-context d_5 can easily be represented in an STG by using term and context non-terminals, where the trick is to locate the forking position of the two holes and use a grammar production with right-hand side of the form $f(\dots, B, \dots, D, \dots)$, where B is a term non-terminal and D a context non-terminal containing the hole of A_2, and B represents the rest of the exceptional P-occurrence, and $val(B) = val(P^{k_2}P_6)$. There are several cases as follows.

(a) $val(S)$ is periodic in the direction $val(P)^\infty$. Let h be such that S exactly matches P^hP_7, such that $holep(S) = holep(P^hP_7)$, where $holep(P_7)$ is a proper prefix of $holep(P)$. If $h \leqslant k+k_1$, then there is a new entry in the result table: $(C, P, k+k_1-h)$. The prefix table has a new entry $(C', P, 1 + k_2)$, in the non-compatible table, corresponding to a sequential overlap, where C' is constructed to represent CP^{k+k_1+1-h}.
In order to find the entries where S overlaps a part of P_6, we split S into $S = S'_0S_1S'_1S_2S'_2 \dots S_mS'_m$, where S_i, S'_i are fresh context non-terminals, $val(S_i) = val(P)^{j_i}$ for some $j_i \geqslant 0$, and $holep(val(S'_i)) = holep(P)$ for all i, S'_i matches P for all i, and

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

$m+1 \leqslant |FV(val(S))|-1$. Here, S'_m is another name for P_7. For every potential entry $CP^{k'}$, such that the position of S'_i is exactly at the start position of P_6 an extra match test has to be made, whether or not it is in the result table, or in the prefix table, and if yes in which prefix table.
Note the number of these tests is at most the number of holes in $val(S)$.

(b) Only a prefix of $val(S)$ is periodic in the direction $val(P)^\infty$, i.e. S can be split into $S = S'_0 S_1 S'_1 S_2 S'_2 \ldots S_m S'_m R$, where S_i, S'_i, R are fresh context non-terminals, $val(S_i) = val(P)^{j_i}$ for some $j_i \geqslant 0$, and $holep(val(S'_i)) = holep(P)$ for all i, S'_i matches P for all i, and $m+1 \leqslant |FV(val(S))|-1$, and m is maximal. Then, there is no periodic entry.
For every potential entry $CP^{k'}$, such that the position of S'_i or R is exactly at the start position of d_5, an extra match test has to be made, whether or not it is in the result table, or in the prefix table, and if yes in which prefix table.
The same for the potential entries $CP^{k'}$, such that the position of S'_i or R is exactly at the start position of P_6, an extra match test has to be made, whether or not it is in the result table, or in the prefix table, and if yes in which prefix table.
Again, the number of these tests is at most twice the number of holes in $val(S)$.

The *final step* is to perform a compaction of the prefix tables for A as follows:

The entry forms are (C) and (C, P, n). First, we look for the entries in a fixed prefix table for A and for hole j: Since the hole path of hole j of $val(S)$ is compatible with the hole path of A for all the entries, it is possible to apply the Periodicity Theorem 3.4 for overlaps with cut.

We scan through the main path of $val(A)$ to check for redundant entries as follows. We start with $c_0 := |holep(A)|$, and make a compaction for values $2^{-i}c \leqslant c_0$ and describe the compaction with c as parameter: Let $c \leqslant c_0$ be a positive integer, let h be the number of holes of $val(S)$. Then, let $d := |holep(A)| - c$ and $e := \lfloor d/(2(h+2)) \rfloor$. Now, we check every interval $c, c+e, c+2e \ldots c+((h+2)-1)e$ whether the number of entries in the interval exceeds $(h+2)$, where we interpret the periodic entries in the interval as if at least $h+2$ single entries are expanded. If this is the case for the interval $[c+ie, c+(i+1)e]$, we can apply the periodicity theorem for multicontexts with h holes and a fixed path p, which is the hole path for hole j of $val(S)$, and generate an entry (C, P). C is determined as the context in $val(A)$ with hole at the first entry in the interval, and P is the context starting there with hole path as given by A, and with a hole depth that is the gcd as computed in Theorem 3.4. The fresh non-terminals C, P are to be constructed by extending the STG. All other entries in the interval are then removed: removing the single entries and by shifting the start of the periodic entries to the right.

The scan then proceeds in the second half of the interval, but with freshly computed c, e. The scan stops if the interval is of length 2.

In case we work on the table for A for the non-compatible entries, the compaction has to perform extra checks: First, two different entries C_1, C_2 are compared for the mchp say q_1, q_2 of $val(S)$ starting at the hole of C_1 (C_2, respectively) with the path of A. If $q_1 = q_2$, then we are in the parallel overlap scenario, otherwise, we are in the sequential overlap scenario.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

In the parallel overlap scenario, the compaction is performed exactly as for the entries in the compatible entries.

In the sequential overlap scenario, first the position q_0 has to be determined, which is the maximal position where all the hole paths of the prefixes deviate from the hole path of A. Therefore, it is sufficient to compute for one entry C_1 with hole position p_1 the mhcp q_1 with $val(A)$. Then, we use q_0 as the right end of the interval, i.e. we compute the overlaps with A' which represents the prefix of $val(A)$ with hole path q_0. Then, the compaction is performed exactly as the other compaction steps. □

Note that all positions in the algorithm have to be compressed by SLPs. Note also that non-terminals are never expanded. Every mention of $val(A)$, $val(P)$ and $val(S)$ is only for being precise, but the computation is always on the STG-level.

It remains to describe an algorithm for the positions of the matchings in term non-terminals using the prefix tables.

Algorithm 3.14 (Construction of the matchings of linear terms with term-non-terminals). Let G be an STG, and let S, T be the non-terminals, such that $val(S)$ is a linear term and we search for submatchings in $val(T)$. Note that we have assumed that G has only productions for term non-terminals of the form $A \to CA'$, where $|val(A')| = 1$, which is possible without much overhead by Proposition 2.12. Assume that the prefix and result tables are constructed as described in Algorithm 3.13. Then, Lemma 2.14 shows that the following tests are sufficient to obtain all positions. Let G_T be the subgrammar that contains all productions for a computation of $val(T)$. The matching positions can be found as follows:

1. Every position in the result table of Algorithm 3.13 is a submatching.
2. For every term non-terminal A of G_T, if S exactly matches A, then a submatching is found.
3. If a non-terminal A has grammar production $A \to CA'$, then we know that $|val(A')| = 1$ and $A' \to a'$ for a constant a'. We use the entries in the prefix tables for C.
   (a) For a single entry C', we determine the compressed position p (the hole position of C'), and then we try to match S at p in CA'.
   (b) For a periodic entry (C', P', n), we have to test whether S exactly matches $(P')^k P_1[a']$ where the names and the tests are similar, but far more restricted, as in the fifth case of Algorithm 3.13. I give some hints: The compatible case is not possible, since such an entry would already be contained in the result table. For the non-compatible overlaps, we have to determine the single critical position that fits, which is possible since periodic positions can be determined from the entry and from S.

3.5. *Properties of the linear Submatching algorithm*

The compaction step is correct, which follows from Theorem 3.4 and Proposition 3.7, and since the construction obeys the assumptions of the theorems.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The potentially dangerous step, which might lead to an explosion (iterated doubling) of entries, is the inheritance of the entries of A_1, A_2 as entries of A with production $A \to A_1A_2$, since there may be several occurrences of A_1, A_2 in other productions. This explosion is prevented on the one hand by placing completed matchings into a result table, and on the other hand by the compaction using a compact representation of periodicities.

We show that the compaction step guarantees a polynomial size of the table(s).

Proposition 3.15. Given an STG G as input, the size of the tables generated in Algorithm 3.13 is $O(|G|^5)$ due to the compaction step.

Proof. The number of tables is at most $O(|G|^2)$, since the number of holes is linear in the size of G and also the number of non-terminals A is at most $|G|$. It remains to show that the number of entries is polynomial. The size, and hence the hole depth of $val(A)$ for non-terminals A is at most $2^{|G|}$, and by the interval division method and the compaction, at most $(h+1) \cdot (h+2) \cdot |G|$ entries will remain in an entry for A, which is of order $O(|G|^3)$. □

The tabling algorithm increases the STG by further non-terminals. We argue that this increase is moderate, i.e. is within the overall polynomial time complexity.

Lemma 3.16. The subtasks of Algorithms 3.13 and 3.14 that increase the STGs that represent the matching positions lead to an at most polynomial size increase, and can be performed in polynomial time.

Proof. The compaction step is responsible for keeping the overall size of the tables at most $O(|G|^5)$. There are several constructions of new non-terminals, for example in the fifth case, like $C, P, P_1, \ldots, P_7$, whose size and depth increases are estimated in Lemma 2.7. The number of these construction operations is polynomial. Since there may be an iterated increase, i.e. the freshly constructed non-terminals may make use of other previously constructed non-terminals, we have to take care of iterated increase. Thanks to Theorem 2.11, also an iterated increase only leads to a polynomial size increase of the final STG. □

As a summary, we have the following theorem.

Theorem 3.17. Let G be an STG, and S, T be two term non-terminals such that $val(S)$ is a linear term, and the submatching positions of $val(S)$ in $val(T)$ are to be determined. Then, Algorithm 3.14 together with Algorithm 3.13 computes a polynomial representation of all submatchings of $val(S)$ in $val(T)$ in polynomial time dependent on the size of G.

4. Submatching algorithms for other cases

In this section, we consider several specialized situations, for example, uncompressed patterns, DAG-compressed terms and also an algorithm for the general case of non-linear terms, which may, however, require exponential time in the worst case.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

4.1. *Ground term submatching*

The investigation in Gascón *et al.* (2008) shows that (exact) term matching, also in the fully compressed version including the computation of a compressed substitution, is polynomial. That is, given two non-terminals S, T, where S may contain variables, there is a polynomial time algorithm for answering the question whether there is some substitution σ such that $\sigma(val(S)) = val(T)$, and also for computing the substitution, where the representation is a list of variable-non-terminal pairs, and the non-terminals belong to an extension of the input STG.

Algorithm 4.1 (Ground compressed term submatching). The special case of submatching where s is ground and compressed by a non-terminal S can be solved in polynomial time by translating both compressed terms into their compressed preorder traversals (i.e. strings) (Busatto *et al.* 2005, 2008), and then applying string pattern matching (see Lifshits 2007; Rytter 2004 for further references on the subject). The translation efficiently computes an SLP for the preorder traversal of $val(S)$ and $val(T)$ and asks whether one is a substring of the other (see Gascón *et al.* 2011 for more information on the preorder traversal). The string matching algorithm in Lifshits (2007) computes a polynomial representation of all occurrences. Note that in our case, the structure of ground terms is very special as a string matching problem: periodic overlaps of the preorder traversal as strings are not possible. Thus, the complete output of the algorithm is a follows: (i) a list of term non-terminals N of the input STG G, where $val(\sigma(S)) = val(N)$, and (ii) a list of pairs (N, p), where the production for N is of the form $N \to CN'$, p is a compressed position, and $val(C)_{|val(p)}[val(N')] = val(S)$. Moreover, every non-terminal N appears at most once in the list.

There are efficient algorithms for the compressed string pattern match (Jez 2012; Lifshits 2007). The required time is $O(n^2m)$, where n is the size of the SLP of T and m is the size of the SLP of S (the pattern). Since the preorder traversal can be computed in linear time (see Gascón *et al.* 2011), we have the following theorem.

Theorem 4.2. Let S, T be non-terminals compressed with STGs G_S, G_T, respectively, such that s is ground. Then, the ground compressed term submatching, whether s occurs in t, can be computed in time $O(|G_T|^2|G_S|)$, and the output is a list of linear size.

Note that there may be exponentially many matching positions, even if the output list has only a single element N, since N may occur at an exponential number of positions in the derivation of $val(G)$.

4.2. *Decision algorithm for uncompressed linear submatching*

Given an uncompressed and linear term s and an STG G together with a non-terminal T, the following algorithm solves the submatching decision problem, but does not compute the substitution: Construct a non-deterministic tree automaton T_s (Comon *et al.* 1997) that recognizes whether s is a subterm of another term ignoring the variables. This automaton can be easily constructed and has a linear number of states, where T_s has an accepting

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

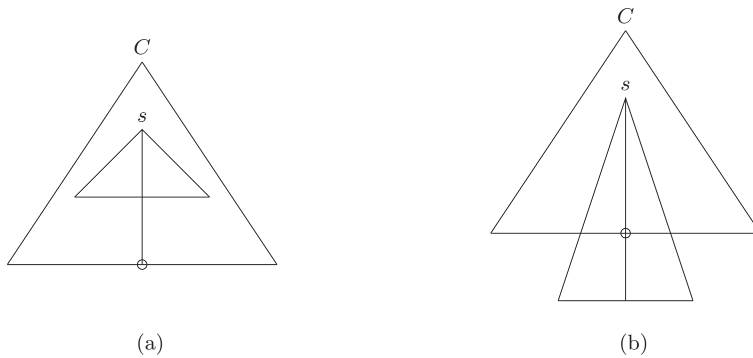

Fig. 3. Cases in the construction of the s-in-C-table.

run on $val(T)$ if and only if s occurs in $val(T)$ as a multicontext (i.e. there is some σ, such that $\sigma(s)$ is a subterm of $val(T)$). It is known that acceptance of a compressed term by nondeterministic tree automata (NTAs) can be decided in time $O(|G|^3 + |G|^2 a^3)$, where a is the size of the automaton (Lohrey and Maneth 2005; Lohrey *et al.* 2009), which means that uncompressed linear submatching can be decided in time $O(|G|^5)$.

This method does not answer further questions like: What is the substitution, all matching positions or matching positions satisfying further constraints. The reason is that this method cannot identify the matching position, since it computes only functions from states to states.

4.3. *DAG-compressed non-linear submatching*

Now, we look for the case of DAG-compressed s, which is slightly more general than the uncompressed case, and where variables may occur several times in s. We show that also in this case, there is an algorithm for submatching that requires polynomial time. The algorithm outputs enough information to determine all the positions of a submatch.

Example 4.3. The number of possible substitutions for a submatch in a DAG-compressed term may be exponential: Let the productions be $S \to f(x, y)$, and $T \to f(A_1, A_1), A_1 \to f(A_2, A_2), \ldots, A_{n-1} \to f(A_n, A_n), A_n \to a$. Then, $val(T)$ is a complete binary tree of depth n and there is a submatch at every non-leaf node. Clearly, it is sufficient to have all A_i as submatchings in the output, which is of linear size.

Algorithm 4.4 (DAG-compressed submatching: overview). We give an overview description of the algorithm for DAG-compressed submatching in the following:

The following task is solved: Let G be an STG, T be a term non-terminal of G, and let G_S be a DAG for the term non-terminal S.

Then, find one (or all) possibilities for: a substitution σ, a term non-terminal S of G_S and a position p such that $\sigma(val(S))$ occurs in $val(T)$ at position p.

We assume that the substitutions are STG-compressed and that the positions are SLP-compressed.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The output will be a list with elements of a kind according to the cases in Lemma 2.14:

1 a term non-terminal N such that s matches N exactly (case (1));
2 a term non-terminal N with production $N = CB$ and a position and a compressed substitution, such that s overlaps C and B (case (2a));
3 a substitution ρ such that $\rho(s)$ is ground, where the positions can be determined using Algorithm 4.1 (case (2(b)i)) with multiple occurrences of the variable x);
4 a context non-terminal C, a substitution ρ, and a compressed position of an occurrence of s in C, where $\sigma(s)$ occurs in $val(C)$, and a variable x with a single occurrence in s, where x matches a superterm of the hole (case (2(b)ii)). □

Definition 4.5 (s-in-C-table, result-list). Let S be a term non-terminal of G_s. The *s-in-C-table* has two coordinates: a context non-terminal C, and a position p that is a suffix of the hole path of $val(C)$ as well as a position of $val(S)$, and there is at most one substitution entry ρ in the s-in-C-table, representing the possibilities where a match of $val(S)$ starts in $val(C)$ at the hole path (see Figure 3). The substitution ρ instantiates certain variables of $val(S)$ with ground terms. We assume that ρ is represented as a list with elements of the form $x \mapsto D_x$, where D_x are non-terminals in an extension of G. The semantics is that $\rho(val(S))$ has an overlap with $val(C)$ at some position q in $val(C)$, and p is a position in $val(S)$, such that $q.p$ is the hole position of $val(C)$.

Entries in the *result list* are as follows:

— (C, p, D), where C is the context non-terminal, p a position in the context on the path $holep(val(C))$, and D a non-terminal representing a ground context that is a prefix of $val(C)_{|p}$.
— s_0, which is a ground instance of s. □

We use a dynamic programming algorithm for a bottom-up (w.r.t. G) computation of the s-in-C-table. In a precomputation several attributes and further information can be computed like an SLP for the position of the holes in $val(C)$ for every context non-terminal C.

The s-in-C-table and the result-list for the non-terminal S of G_S is as follows. Let $s = val(S)$ in the following.

Algorithm 4.6 (For DAG-compressed term, construction of the s-in-C-table, result table and all submatchings). We assume given an STG G, a fixed non-terminal S, a context non-terminal C.

— The production is $C \to f(A_1, \ldots, A_{j-1}, [\cdot], A_{j+1}, \ldots A_n)$. Then, compute a compressed substitution ρ, such that for $S \to f(S_1, \ldots, S_n)$, $val(\rho(S_i)) = val(A_i)$ for all $i = 1, \ldots, n$, where $i \neq j$. If there is no such ρ, then there are no entries.
Let ρ be the computed substitution.
If $val(S_j) = x$ and x occurs only once in s, then there is no s-in-C-table entry, and we are in case (2(b)ii) of Lemma 2.14, and the entry in the result-list will be $(C, \varepsilon, \rho(s[[\cdot]/x]))$.
Otherwise, there will be the table entry ρ for (C, j).

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

— The production is $C \to C_1C_2$, and an entry in the table is to be computed from the entries in the tables for C_1, C_2. The entries in the table for C_2 are simply inherited to the table for C.
If there is an entry ρ for (C_1, p) in the table for C_1, then we have to match $s_{|p}$ against C_2. There are several cases as follows:

1 The hole position q_2 of $val(C_2)$ is a position in $s_{|p}$. Then, we compute a compressed substitution ρ' extending G as follows: For the context non-terminal D with $val(D) := s_{|p}[[\cdot]/q_2]$, the substitution is computed as an exact match such that $\rho'(val(D)) = val(C_2)$.
If there is no such ρ', then there will be no entries.
If ρ' and ρ can be joined, then let the ρ'' be the combined substitution, otherwise there is no entry in the table.
If $s_{|p.q_2}$ is a variable that occurs exactly once in s, then the entry in the result-list is (C, q', D'), where D' is a context non-terminal representing $\rho''(d)$, and q' is such that $q'p = holep(val(C_1))$.
Otherwise, ρ'' is the new entry for $(C, p.q_2)$.

2 The hole position q_2 of $val(C_2)$ is not a position in $s_{|p}$. Then, let q be the maximal prefix of $p.q_2$ that is also a position in s. Compute ρ' for the match of the non-terminal D with C_2, where $val(D) = s_{|p}[[\cdot]/q]$, i.e. such that $\rho'(val(D)) = val(C_2)$.
If ρ' does not exist, or if it is computable but not joinable with ρ, then there is no entry.
Let ρ'' be the join of ρ and ρ'. Clearly, $s_{|q}$ is now a variable, say x. If x occurs exactly once in s (case (2(b)ii)) of Lemma 2.14), then the entry in the result-list is $(C, p.q_2, D')$, where D' represents $\rho''(val(D))$.
Otherwise (case (2(b)i)) of Lemma 2.14) $\rho''(s_{|q})$ is a ground term. Then, the entry in the result-list is simply the compressed substitution ρ'', such that $\rho''(s)$ is a ground term.

The representation of all occurrences of submatchings of S in non-terminals of G is as follows:

1 The representation of all occurrences of ground terms in the result list are computed using Algorithm 4.1.
2 The entries (C, p, D) in the result-list are kept.
3 For all term non-terminals N of G, check whether there is a substitution σ, such that $\sigma(s) = val(N)$. If this is true, then a match is found.
4 For all non-terminals N with production $N \to CB$, and every position p of s that is also a suffix of $holep(C)$, compute a compressed match ρ' for $s_{|p}$ with B. If ρ' and the s-in-C-table entry ρ for p are compatible, then a match is found, where the substitution is ρ'', the join of ρ and ρ'. □

Proposition 4.7. Algorithm 4.6 for submatching in the case that s is uncompressed or DAG-compressed has polynomial running time in $|G_s|$ and the size of G. Moreover, a polynomial sized representation of all matching possibilities can be computed.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The number of necessary matching substitutions is polynomial, if the subcontext case, i.e. case (2(b)ii) of Lemma 2.14 is represented as a partial substitution, as done in Algorithm 4.6

Proof. The table is of at most quadratic size, and the number of entries is at most linear. The number of potential positions appearing in the table is the set of non-trivial positions p in s, which are suffixes of the hole path of $val(C)$ of the context non-terminal C. For fixed C, upper bound for this number is $depth(s)$. However, the non-terminals used in the substitutions ρ have to be constructed in the STG G'. This constructions can be done independently of each other. The join of (compressed) substitutions requires an equality comparison of compressed terms, which can be performed in polynomial time (Busatto *et al.* 2005). Hence, the size of the table, and of the output, is polynomial. □

Note that the number of submatching positions may be exponential, since for example a matching non-terminal N may have an exponential number of occurrences in the input term $val(T)$, but those can be represented in polynomial space. Also, the number of substitutions may be exponential, if all ρ are requested such that $\rho(s)$ is a subterm of t. Note that we avoid the construction of substitutions in the cases where $s = d[x]$, x occurs exactly once in s and d occurs as a subcontext of some context $val(C)$, where C is a context non-terminal.

As a summary, we have the following theorem.

Theorem 4.8. Let G_s, G_t be a STGs, S, T be two term non-terminals in G_s, G_t, respectively, where G_s is a DAG. Then, the submatch computation problem can be solved in polynomial time. Also, an explicit polynomial representation of all matching possibilities can be computed in polynomial time.

5. Submatching in the non-linear STG-compressed case

A non-deterministic polynomial (NP) time submatching algorithm is given in the general case, where s as well as t are STG-compressed and a submatching of s in t is requested. Also, a deterministic polynomial time submatching algorithm is described if there are a few variable occurrences.

5.1. *A non-deterministic algorithm for sub-matching in the general case*

We describe a simple and sufficiently efficient non-deterministic algorithm, which, after making it deterministic, may require exponential time in the worst case.

Given an STG G, two term non-terminals S, T, where $val(S)$ may contain variables, compute an extension G' of G and a compressed substitution σ such that $\sigma(val_{G'}(S))$ is a subterm of $val_G(T)$. Also, representations of the position(s) of the match in $val(T)$ have to be computed.

Algorithm 5.1 (Non-deterministic submatching for non-linear terms). Assume given non-terminals S, T with $s = val(S), t = val(T)$. The (non-deterministic) algorithm for matching s against a subterm of t proceeds in several steps: The algorithm consists of an iteration

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

that in every step (non-deterministically) generates instantiations for at least one variable of $val(S)$ and extends G. The iteration is described as a single step with input (G, S), and constructing (G', S') for the next iteration step, and also collecting instantiations. Note that G_T, which is the part of the STG that generates T, is unchanged during the iteration. There are following two cases:

1 If s contains a variable with more than two occurrences, then do the following: Construct an extension G_1 of G and non-terminals S_1, C, A such that $S_1 \to CA$ is a production, $val(S_1) = s$, $val(C)$ is a linear context, $val(A) = f(r_1, \ldots, r_n)$, $n \geqslant 2$ and there is a variable that occurs in at least two r_i. This prefix construction is a slight variation of the prefix construction in Lemma 2.7 with an extra test for variable-occurrences. Then, (non-deterministically) select a right-hand side $f(B_1, \ldots, B_n)$ of a production in G_T contributing to T.
The first case is that $val(f(B_1, \ldots, B_n))$ is a term. Then, compute σ as matching A against $f(B_1, \ldots, B_n)$ (see Gascón *et al.* (2008, 2011)). If it fails, then there is no result in this path of the non-deterministic computation. If it succeeds, then it means to construct an extension G_2 of G_1 such that $val_{G_2}(A) = val_G(f(B_1, \ldots, B_n))$. Note that G_2 contains the instantiations for the variables occurring in $val(A)$. Now, $val_{G_2}(A)$ is ground. The next step of the iteration with $G' := G_2$ and $S' := S_1$ will use the item for the linear case.
The second case is that $f(B_1, \ldots, B_n)$ is a context, and where i is the hole position. Then, construct context non-terminals $A_1, \ldots, A_n$, such that $val(A) = val(f(A_1, \ldots, A_n))$ and add them to the STG G giving G_2. Then, compute σ as matching $f(A_1, \ldots, A_n)$ against $f(B_1, \ldots, B_{i-1}, [.], B_{i+1}, \ldots, B_n)$ ignoring the index i, which is the hole position. In the case of a successful match, this means to construct an extension G_3 of G_2, matching A_j against B_j for all $j \neq i$. The extension G_3 contains the instantiations for the variables occurring in A_j such that $val_{G_3}(A_j) = val_G(B_j)$. Now, $val_{G_3}(S_1)$ contains less variables than s and we can perform the next iteration step with $G' := G_3$, and $S' := S_1$.

2 If s is linear, then Algorithm 3.13 is applied for the now linear compressed term s and the compressed ground term t.

Theorem 5.2 (Non-deterministic general submatch). Let G be an STG and let S, T be two non-terminals, and $s = val(S), t = val(T)$, where s may contain variables. Then, the algorithm for fully compressed submatch for compressed terms s, t requires at most searching in $|G|^{|FVmult(s)|}$ alternatives for the substitution and the computation for one alternative can be done in polynomial time. Thus, the submatching problem is in NP.

Proof. A single computation path of Algorithm 5.1 can be completed in polynomial time, which follows, since in every iteration step at least one variable will be instantiated, and the overall increase of the STGs is at most polynomial due to Lemma 2.7. The time for the subalgorithms is polynomial, see Theorems 4.2 and 3.17. Thus, the submatching decision problem is in NP. □

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

5.2. *Non-linear compressed submatching*

In the general submatching case, if there are a large number of variable occurrences in s, then the method to first linearize terms and then to post-process the result is prohibitive, since even comparing two linear STG-compressed terms for equality (if variables are represented in the STG using a single symbol for a hole, and every hole is implicitly a different variable) is coNP-hard (M. Lohrey, personal communication, 2013), and since such equality comparisons were required in the linear submatching computations. However, if the number of occurrences of variables in s is small, then this method can be applied, as we will show.

The non-linear submatching problem can be computed as follows.

Algorithm 5.3. Given the non-terminal S, where $s = val(S)$ is a non-linear term, first linearize s as follows: construct a non-terminal S_{lin} from S such that $s_{lin} := val(S_{lin})$, the linearized term s, and solve the submatching problem for S_{lin} and generate the result tables for the term or context non-terminals A. This results in polynomial-sized result-tables with single and periodic entries and submatchings from the term-non-terminal Algorithms 3.13 and 3.14.

Then, check consistency of the entries with the variable structure of s as follows:

1 Let C be a single entry in the result table of the context non-terminal A. Note that $val(C)$ is a prefix of $val(A)$ such that also $val(C)(val(S_{lin}))$ is a prefix of $val(A)$. Now, fix this position of s in $val(A)$.
For all variables x in s, where x has an occurrence not on the hole path of $val(A)$: compute the instantiation for x. After an exhaustive instantiation, $\sigma(s)$ is either ground or contains a single variable occurrence. Then, compute the submatchings of $\sigma(s)$ using the known methods. This can be done in polynomial time.

2 Check consistency of the periodic entries:
Consider the first and last $h + 1$ subentries as single entries using the previous item.
Now, consider the other subentries. Note that the periodicity Theorem 3.4 for full overlaps shows that for every variable x of s_{lin}, the instantiation of x for all the occurrences of s_{lin} in the remaining periodic sequence are equal. Hence, it is sufficient to consider a single one. So, check the first instance subentry like a single entry, which will produce a representation of submatches of the instance of s.

The submatches in the term non-terminals obtained by first linearizing s, applying Algorithm 3.14, and then post-processing the solutions is done in a similar way as for context non-terminals. □

Theorem 5.4. Let S, T be non-terminals with $s = val(S)$, and where $Occ_S(V)$ is the number of occurrences of variables in $val(S)$. Then, a representation of all submatches of s in $val(T)$ can be computed in time polynomial in $|G|$ and $Occ_S(V)$. In particular, if the number of occurrences of variables in s is not greater than $|G|$, then submatching can be performed in polynomial time in $|G|$.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Proof. This follows by using Algorithm 5.3, Theorem 3.17, the methods for succinctly constructing extensions of the STG and estimating the size and computation time. □

Note that Theorem 5.4 does not lead to a polynomial algorithm for submatching and rewriting if the number of occurrences of variables in the pattern term is too large.

6. Polynomial compressed term rewriting

In this section, we apply the results on the compressed submatch to sequences of reductions by a TRS, which is also applicable to an equational deduction step.

Given a TRS R and a term t, where we assume that the rewrite rules of R as well as t are compressed by STGs, we investigate upper bounds for execution time and the growth of the STG. There are several options for a rewriting strategy, i.e. for choosing the position(s) for a deduction step: (i) single-position rewritings at leftmost-innermost or outermost-leftmost or any positions, and also (ii) parallel rewritings of the same subterm at several positions using the same rewrite rule. For our compressed representation the natural approach is to use parallel rewriting of the same subterm at several positions and by the same rewriting rule. Note, however, that the set of redexes that are rewritten in parallel will depend on the structure of the grammar, and not on the structure of the rewritten term. We do not investigate rewriting strategies and/or consequences for computing normal forms. If the rewriting system is confluent, then it is correct to use (opportunistic) parallel rewriting, which is likely to have less rewriting steps to a normal form.

First, we define how a (parallel or single-position) rewriting step including the computation of the redex on an STG-compressed term t is performed.

Algorithm 6.1 (Compressed rewriting step with an oracle for the redex). Assume given a compressed TRS R and a ground term t compressed with non-terminal T and STG G with $val(T) = t$, where we assume that R is compressed by the STG G_R as $\{L_i \to R_i \mid i = 1,\dots,n\}$ and L_i, R_i are term non-terminals.

A term rewriting step is performed as follows.

First select $L_i \to R_i$ as the rule. There is an oracle, which is one of our submatching algorithms applied to L_i, for finding the redex for $val(L_i)$ or the set of redexes that provides the following:

1. An extension G' of G, i.e. additional non-terminals and productions.
2. A substitution σ as a list of pairs: $\{x_1 \mapsto A_1,\dots,x_m \mapsto A_m\}$, where $FV(val(L_i)) = \{x_1,\dots,x_m\}$, A_i are term non-terminals in G', and $val(A_i)$ is a subterm of t. It is also assumed that the instantiation is integrated in the grammar G' as productions $x_i \to A_i$ for $i = 1,\dots,m$.
3. A term non-terminal A in G' that contributes to $val(T)$, and a compressed position p.

The size-assumption is that G' is G extended only for the non-terminals A_i representing subterms of t, and the size of the grammar for p is linear in $|G|$. We also assume that for several rule applications, every rewrite step uses a fresh copy of the STG G_R.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

The output will be an STG G'' constructed from $G' \cup G_R$ using one of the cases below.

The oracle will be one of the submatch algorithms above, and act non-deterministically: First, it computes a submatch of some left-hand side L_i of a rewrite rule $L_i \to R_i$ of R in $val(T)$. This will lead to a grammar G' that is an extension of $G \cup G'_R$, where G'_R is a copy of G_R, and to the representation of the position(s) and substitution of the submatch. This holds, for example, for Algorithms 3.13, 5.1 and 4.4. Then, one of the parallel redexes is selected. There are following different cases:

1. If the position is trivial, i.e. $p = \varepsilon$, then $val(A) = \sigma(val(L_i))$, and the operation is to modify $G' \cup G_R$ by replacing the production for A by $A \to R_i$.
2. The other case is that the production for A is $A \to D[A']$ and the position p points to a suffix of $val(D)$, i.e. the match of L_i is in the middle between A and A' in $val(D[A'])$. Starting with $G' \cup G_R$, and using the returned (compressed) position, we construct context non-terminals D_1, D_2, including their corresponding productions, such that $val(D_1D_2) = val(D)$ and $val(D_2[A]) = \sigma(val(L_i))$. The rewrite step is then performed by replacing the production $A \to D[A']$ by $A \to D_1[R_i]$.
3. In case a single-position rewrite step instead of a parallel is requested, only the construction of the non-terminal corresponding to the position has to be changed. The overall complexity estimations are not changed by this modification.
4. There is an alternative treatment of the rewriting step in item (2) in the exceptional case that the rewrite rule can rewrite a context into another context, which is potentially a very efficient rewriting step in our representation.
   Assume given the term non-terminal A with production $A \to D[A']$ and the position p that points to a suffix of $val(D)$. The condition for application is that the term $val(L_i)$ has exactly one occurrence of the variable x_1, $val(R_i)$ has at most one occurrence of the variable x_1 and $holep(val(D)_{|p})$ has q as prefix, where q is the position of x_1 in $val(L_i)$.
   Let $\sigma_1 := \{x_2 \mapsto A_2, \ldots, x_m \mapsto A_m\}$, where $\sigma := \{x_1 \mapsto A_1, \ldots, x_m \mapsto A_m\}$.
   We start with $G' \cup G_R$. Then, we remove the instantiation of x_1 and turn L_i into a context non-terminal, by adding $x_1 \to [\cdot]$. Construct productions as follows: Let D_1, D_2, D_3 be fresh context non-terminals with productions, such that $val(D) = val(D_1D_2), val(D_2) = val(L_iD_3)$, and p represents the hole path of $val(D_1)$.
   Then, the exceptional rewriting can take place:
   If x_1 is not contained in $val(R_i)$, then the rewriting step turns D into a term non-terminal with production $D \to D_1[R_i]$, and also adjusts the grammar bottom-up by turning several context non-terminals into term non-terminals.
   If x_1 is contained in $val(R_i)$, then the rewriting step is to replace the production for D by $D \to D_1D_2$ and the production $D_2 \to L_iD_3$ by $D_2 \to R_iD_3$. □

Now, we estimate the size increase of the STG that is used to represent t_n, which is the final term after n rewrite steps, i.e. $t \to_R \ldots \to_R t_n$. We distinguish between the STG G_R for the rewrite rules and the STG G_n for compression of t_n. Repeating the term rewriting step is done by using a fresh copy of G_R, the STG compressing the TRS R.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

In order to complement the estimations in Lemma 2.7, we have to check the size increase by constructing the substitution, and the estimations for the size increase by the rewrite step itself.

Lemma 6.2. A rewrite step according to Algorithm 6.1 where the submatching itself is seen as a free oracle, i.e. is not counted, satisfies the following estimations:

$$\begin{array}{lll} Vdepth(G',V') & \leqslant & Vdepth(G,V) + \log(depth(G)) + depth(G_R), \\ |V'| & \leqslant & |V| + M, \\ |G'| & \leqslant & |G| + M depth(G) + M + depth(G)^2 + |G_R|, \end{array}$$

where $M = \max_i(|FV(r_i, l_i)|)$.

Proof. Let $L_i \rightarrow R_i$ be the selected rewrite rule.
The contributions are: (i) For every variable $x \in FV(val(L_i))$: its substitution construction, i.e. constructing $|FV(val(L_i))|$ times a subterm of t, and then making an instantiation. (ii) Rearranging G such that the position of the matched subterm $val(\sigma(L_i))$ is explicit and (iii) modifying the productions of G by replacing L_i by R_i.

The construction (i) is independent from (ii) and (iii). Constructing the substitution consists in $|FV(val(L_i))|$ times independently constructing a subterm of t, which increases the size by $M \cdot depth(G)$, the $Vdepth$ only by $\log(depth(G))$ (since we can take the maximum) and by instantiating, which adds M productions, adds M variables to V, but does not change the $Vdepth$. The replacement of L_i by R_i may increase the $depth$ and $Vdepth$ at most by $depth(G_R)$. Rearranging requires to construct a prefix-context of a term and changing a production. This may add $depth(G)^2$ non-terminals for the prefix and the non-terminal L_i plus its definition, which means to add $|R|$ to the size. □

Lemma 6.3. Let there be a sequence $G_i, i = 0, \ldots, n$ of STGs generated by extension or transformation and sets $V_i, i = 0, \ldots, n$ of instantiated non-terminals, such that the following holds:

$$\begin{array}{lll} Vdepth(G_i,V_i) & \leqslant & Vdepth(G_{i-1},V_{i-1}) + \log(depth(G_{i-1})) + r, \\ |V_i| & \leqslant & |V_0| + r \cdot i, \\ |G_i| & \leqslant & |G_{i-1}| + (depth(G_{i-1}) + r)^2, \end{array}$$

where we assume $|V_i| < r$ and $2 \leqslant r$.
Then, $|G_n|$ is bounded by $\mathcal{O}\big(r^2 n^7 \big(|G_0|^2 + |G_0|(\log n + 2r) + (\log n + 2r)^2\big)\big)$.

Proof. From Lemma 2.8, we derive $depth(G_i) \leqslant i \cdot 2r \cdot Vdepth(G_i, V_i)$. Therefore, the recurrence for $Vdepth(G_i, V_i)$ may be replaced by

$$Vdepth(G_{i+1}, V_{i+1}) \leqslant Vdepth(G_i, V_i) + \log Vdepth(G_i, V_i) + \log i + 2r. \tag{6.1}$$

Let n be the final index i.

A first bound for these recurrence can be computed relaxing the inequality as $Vdepth(G_{i+1}, V_{i+1}) \leqslant 2\, Vdepth(G_i, V_i) + \log i + 2r$ that has as solution $Vdepth(G_i, V_i) \leqslant$

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

$2^i\left(Vdepth(G_0,V_0)+n(\log n+2r)\right)$. Replacing this approximate solution in (6.1) results in

$$\begin{aligned} Vdepth(G_{i+1},V_{i+1}) &\leqslant Vdepth(G_i,V_i)+i\cdot\log\left(Vdepth(G_0,V_0)+n(\log n+2r)\right) \\ &\qquad +n(\log n+2r). \\ &\leqslant Vdepth(G_i,V_i)+i\cdot a_1+a_2, \end{aligned} \tag{6.2}$$

where $a_1=\log Vdepth(G_0,V_0)$ and $a_2=2n(\log n+2r)$.

Now, we get the approximate solution

$$Vdepth(G_i,V_i)\leqslant Vdepth(G_0,V_0)+a_1\mathcal{O}(i^2)+a_2 i.$$

Therefore,

$$\begin{aligned} depth(G_i) &\leqslant i\cdot 2r\cdot\left(Vdepth(G_0,V_0)+a_1\mathcal{O}(i^2)+a_2 i\right) \\ &\leqslant 2r\cdot\left(i\cdot Vdepth(G_0,V_0)+a_1\mathcal{O}(i^3)+a_2\cdot i^2\right) \\ &\leqslant 2r\cdot\left(|G_0|\cdot\mathcal{O}(i^3)+a_2 i^2\right). \end{aligned}$$

Replacing this in the recursion for $|G_i|$, we get

$$\begin{aligned} |G_{i+1}| &\leqslant |G_i|+(depth(G_i)+r)^2 \\ &\leqslant |G_i|+r^2(|G_0|\cdot\mathcal{O}(i^3)+2a_2 i^2)^2. \end{aligned}$$

Thus, $|G_n|$ is of order $r^2\left(\mathcal{O}(|G_0|^2n^7)+\mathcal{O}(|G_0|a_2n^6)+\mathcal{O}(a_2^2n^5)\right)$.

Expanding $a_2=n(\log n+2r)$, we obtain the following:
$|G_n|$ is bounded by $\mathcal{O}\left(r^2n^7\left(|G_0|^2+|G_0|(\log n+2r)+(\log n+2r)^2\right)\right)$. □

Theorem 6.4. Let R be a TRS compressed with G_R and t be a term compressed with an STG G. Then, a sequence of n term rewriting steps according to Algorithm 6.1, where submatching is a non-deterministic oracle that is not counted, can be performed in polynomial time. The size increase of the STG by n term rewriting steps is $\mathcal{O}\left(|G_R|^2n^7\left(|G|^2+|G|(\log n+2|G_R|)+(\log n+|G_R|)^2\right)\right)$.

Proof. This follows by combining Lemmas 6.2 and 6.3 using the following estimations: $r:=|G_R|$, $depth(G)\leqslant r$, $|V_i|\leqslant r$, $M\leqslant r$ and $a^2+ar+2r\leqslant(a+r)^2$ for $1\leqslant a$ and $1\leqslant r$, then it is sufficient to solve the recurrent inequations under the assumption $|V_i|\leqslant r$:

$$\begin{aligned} Vdepth(G_i,V_i) &\leqslant Vdepth(G_{i-1},V_{i-1})+(\log(depth(G_{i-1}))+r, \\ |V_i| &\leqslant |V_0|+r\cdot i, \\ |G_i| &\leqslant |G_{i-1}|+(depth(G_{i-1})+r)^2. \end{aligned}$$

The time estimation also holds, since every construction step can be performed in polynomial time and the total size of the final grammar is polynomial. □

Looking at the dependency on the different input sizes, and assuming the others are constant, correspondingly. We have $\mathcal{O}(n^7\log^2(n))$ if only the dependency on the number n of rewrites is of interest; $\mathcal{O}(|G_0|^2)$ is the dependency on the size of the grammar of the reduced term; and $\mathcal{O}(|G_R|^4)$ is the dependency on the size of the rewrite system.

Note that the degree of the polynomial for the estimation of the worst case running time is worse than the space bound. The term rewriting sequence has to be constructed

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

(+1) and Plandowski equality check has to be used in every construction step, which contributes a factor of 3 in the exponent. But, note that there are faster deterministic tests (Jez 2012; Lifshits 2007) and even faster randomized equality checks (Berman *et al.* 2002; Gasieniec *et al.* 1996b; Schmidt-Schauß and Schnitger 2012).

6.1. *Combinations of the results*

We combine the results on submatching and sequences of rewriting. The understanding is that Algorithm 6.1 is used by selecting some of the rewrite rules of the given TRS R and some resulting submatch produced by the appropriate algorithm given above. We obtain the following combination results.

Corollary 6.5. Let R be an STG-compressed TRS and t be an STG-compressed term. Then, a sequence of n term rewriting steps using submatching Algorithm 5.1 can be performed in non-deterministic polynomial time.

Proof. This follows from Theorems 6.4 and 5.2. □

Corollary 6.6. Let R be a left-linear STG-compressed TRS and t be an STG-compressed term. Then, a sequence of n term rewriting steps using the submatching Algorithms 3.13 and 3.14 can be performed in polynomial time.

Proof. This follows from Theorems 6.4 and 3.17. □

Corollary 6.7. Let R be a TRS with DAG-compressed left-hand sides and STG-compressed right-hand sides and let t be an STG-compressed term. Then, a sequence of n term rewriting steps using the submatching algorithm 4.6 can be performed in polynomial time in n.

Proof. This follows from Theorems 6.4 and 4.8. □

Corollary 6.8. Let R be an STG-compressed TRS and t be an STG-compressed term, such that every left-hand side of every rule in R has at most $|G|$ occurrences of variables. Then, a sequence of n term rewriting steps using the submatching algorithm 5.3 can be performed in polynomial time in n.

Proof. This follows from Theorems 6.4 and 5.4. □

6.2. *Relation to results on runtime complexity*

We exhibit a relation of our results to the results on polynomial runtime complexity (Avanzini and Moser 2010); for further information, see also Lago and Martini (2009, 2012).

Let the function symbols be partitioned into defined symbols F_D and constructor symbols F_C. Terms constructed only of symbols from F_C and variables are called *values*. Then, a TRS is called a *constructor-TRS*, if every rewrite rule is of the form $f(s_1,\dots,s_n) \to r$, where f is a defined symbol and s_i are values; the terms $f(s_1,\dots,s_n)$ are also called *basic terms*. Given a constructor-TRS R, its *runtime complexity* $rc_R(n)$ is defined as the maximum length of an R-reduction sequence for all basic terms t with $|t| \leqslant n$. Confluence of a TRS

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

R is defined as usual: for all t_1, t_2, t_3 with $t_2 \ {}_R\!\overset{*}{\leftarrow}\ t_1 \overset{*}{\rightarrow}_R t_3$ there is a term t_4 with $t_2 \overset{*}{\rightarrow}_R t_4 \ {}_R\!\overset{*}{\leftarrow}\ t_3$. A term t_0 is a *normal-form* of a term t w.r.t. a TRS R if $t \overset{*}{\rightarrow}_R t_0$, and t_0 is R-irreducible. Note that if a TRS R is confluent, then R-normal-forms are unique.

Since rewriting using DAG-compression is never forced to introduce contexts, and thus all terms in the sequence remain DAG-compressed, we obtain the following corollary, which re-establishes a result in Avanzini and Moser (2010); however, in a slightly more general form.

Corollary 6.9. Let R be a constructor TRS with polynomial runtime complexity. Then, a sequence of n term rewriting steps, using DAG-compression, can be performed in polynomial time in n. In particular, if the TRS is confluent in addition, the computation of (compressed) normal-forms can be done in polynomial time.

Proof. Given a basic term t with $|t| \leqslant n$, the number of single-position rewrite steps w.r.t. R is also polynomial. Hence, the corollary follows from Theorems 6.4 and 4.8 and Corollary 6.7. □

6.3. *Possible extensions*

Main Theorem 6.4 tells us that the term rewriting algorithm on STG-compressed terms will produce a polynomial-sized STG depending on the number of steps. The following example will show that even for well-behaved equational rules an exponential number of rewrites may be necessary to obtain an irreducible term. The example also shows that an extension of STGs to linear SLCF tree grammars as in Busatto *et al.* (2008); Lohrey *et al.* (2009), i.e. tree grammars using contexts with multiple holes, where every hole occurs once, does not help. Let the TRS be $f(x) \to g(x, x)$, and let the term $f^{2^n}(a)$ be represented as $C_1 \to f(\cdot)$, $C_2 = C_1C_1$, $C_3 = C_2C_2$, $\ldots C_{n+1} = C_nC_n$, $T \to C_{n+1}(x)$. A term rewriting step on T using $f(x) \to g(x, x)$ that is applied to C_1 would produce $C_1 := g(.,.)$, which is syntactically not permitted, since there would be two holes. A complete rewriting sequence of $f^{2^n}(a)$ would produce a binary tree t' of depth 2^n with 2^{2^n} leaves. Since an STG G can produce only terms of size $2^{|G|}$, the STG for t' would be of exponential size. The result of the rewrite could be easily expressed in a non-linear SLCF tree grammar (Busatto *et al.* 2008); however, the complexity of the equality check is currently only known to be in PSPACE, and also the complexity of all other operations like matching, extensions, rewriting etc. has not been investigated yet.

For compressed ground terms and DAG-compressed reduced ground rewrite rules (Schmidt-Schauß *et al.* 2011) shows that normal-forms can be computed in polynomial time, although there may be an exponential number of rewrites for the uncompressed term.

A further topic to be investigated are rewriting strategies like innermost-leftmost or leftmost-outermost asf. in connection with STG-compression.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

7. Conclusion and future research

We have shown that finding an instance of a linear term s as a subterm of t under STG-grammar compression can be done in polynomial time in the size of the STG. We have also shown that the general case of a non-linear term s is in NP, and a representation of all submatches can be computed in time $O(n^{O(|FVmult(s)|)})$. As an application to rewriting, we have shown that a sequence of n (single-position and parallel) rewrites of a compressed term can be computed in polynomial time in n. We plan an implementation of a selection of the matching algorithms to assess their performance in practice.

A remaining open question is whether the general compressed submatching, in particular the case of a non-linear pattern term s with any number of variable occurrences, can be solved in polynomial time or not, even in the case of partial words (Schmidt-Schauß 2012).

Further research is to optimize the submatchings and the rewriting algorithms and to investigate rewriting strategies under compression. Also, the exact complexity of the different cases of submatching may be a subject of future research.

Acknowledgements

I thank David Sabel for discussions and helpful comments. I thank the referees for their remarks which improved the paper. I also thank Rachid Echahed for his help and encouragements.

References

Avanzini, M. and Moser, G. (2010). Closing the gap between runtime complexity and polytime computability. In: Lynch, C. (ed.) *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA '10)*, Leibnitz International Proceedings in Informatics, vol. 6, Schloss Dagstuhl, Germany 33–48.

Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*, Cambridge University Press, New York, NY, USA.

Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W. and Rytter, W. (2002). On the complexity of pattern matching for highly compressed two-dimensional texts. *Journal of Computer and System Sciences* **65** (2) 332–350.

Busatto, G., Lohrey, M. and Maneth, S. (2005). Efficient memory representation of XML documents. In: *Proceedings of the International Workshop on Database Programming Languages (DBPL 2005)*, Lecture Notes in Computer Science, vol. 3774, Springer 199–216.

Busatto, G., Lohrey, M. and Maneth, S. (2008). Efficient memory representation of XML document trees. *Information Systems* **33** (4–5) 456–474.

Comon, H. (1995). On unification of terms with integer exponents. *Mathematical Systems Theory*, **28** (1) 67–883.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M. (1997). Tree automata techniques and applications. release October 2002. Available at http://tata.gforge.inria.fr/

Gascón, A., Godoy, G. and Schmidt-Schauß, M. (2008). Context matching for compressed terms. In: *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, IEEE Computer Society 93–102.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Gascón, A., Godoy, G. and Schmidt-Schauß, M. (2011). Unification and matching on compressed terms. *ACM Transactions on Computational Logic* **12** (4) 26:1–26:37.

Gasieniec, L., Karpinski, M., Plandowski, W. and Rytter, W. (1996a). Efficient algorithms for Lempel–Ziv encoding (extended abstract). In: Karlsson, R. G. and Lingas, A. (eds.) *Proceedings of the SWAT*, Lecture Notes in Computer Science, vol. 1097, Springer 392–403.

Gasieniec, L., Karpinski, M., Plandowski, W. and Rytter, W. (1996b). Randomized efficient algorithms for compressed strings: The finger-print approach (extended abstract). In: *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM' 96)*, Lecture Notes in Computer Science, vol. 1075, Springer 39–49.

Hermann, M. and Galbavý, R. (1997). Unification of infinite sets of terms schematized by primal grammars. *Theoretical Computer Science* **176** (1–2) 111–158.

Jez, A. (2012). Faster fully compressed pattern matching by recompression. In: *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP), Part I*, Lecture Notes in Computer Science, vol. 7391, Springer 533–544.

Karpinski, M., Rytter, W. and Shinohara, A. (1995). Pattern-matching for strings with short description. In: *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM '95)*, Lecture Notes in Computer Science, vol. 937, Springer-Verlag 205–214.

Lago, U.D. and Martini, S. (2009). Derivational complexity is an invariant cost model. In: *Proceedings of the FOPARA*, Lecture Notes in Computer Science, vol. 6324, Springer 100–113.

Lago, U.D. and Martini, S. (2012). On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science* **8** (3:12) 1–27.

Levy, J., Schmidt-Schauß, M. and Villaret, M. (2006). Bounded second-order unification is NP-complete. In: *Proceedings of the 17th International Conference on Term Rewriting and Applications (RTA)*, Lecture Notes in Computer Science, vol. 4098, Springer 400–414.

Levy, J., Schmidt-Schauß, M. and Villaret, M. (2008). The complexity of monadic second-order unification. *SIAM Journal of Computing* **38** (3) 1113–1140.

Levy, J., Schmidt-Schauß, M. and Villaret, M. (2011). On the complexity of bounded second-order unification and stratified context unification. *Logic Journal of the IGPL* **19** (6) 763–789.

Lifshits, Y. (2007). Processing compressed texts: A tractability border. In: *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, Lecture Notes in Computer Science, vol. 4580, Springer, 228–240.

Lohrey, M. (2012). Algorithmics on SLP-compressed strings. A survey. *Groups Complexity Cryptology* **4** (2) 241–299.

Lohrey, M. and Maneth, S. (2005). The complexity of tree automata and XPath on grammar-compressed trees. In: *Proceedings of the 10th International Conference on Implementation and Application of Automata (CIAA'05)* 225–237.

Lohrey, M., Maneth, S. and Mennicke, R. (2011). Tree structure compression with RePair. In: *2011 Data Compression Conference (DCC 2011)*, March 29–31, 2011, Snowbird, UT, USA, IEEE Computer Society 2011, 353–362.

Lohrey, M., Maneth, S. and Schmidt-Schauß, M. (2009). Parameter reduction in grammar-compressed trees. In: *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, Lecture Notes in Computer Science, vol. 5504, Springer 212–226.

Lohrey, M., Maneth, S. and Schmidt-Schauß, M. (2012). Parameter reduction and automata evaluation for grammar-compressed trees. *Journal of Computer and System Sciences* **78** (5) 1651–1669.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press

Plandowski, W. (1994). Testing equivalence of morphisms in context-free languages. In: *Proceedings of the Annual European Symposium on Algorithms (ESA '94)*, Lecture Notes in Computer Science, vol. 855, Springer 460–470.

Plandowski, W. and Rytter, W. (1999). Complexity of language recognition problems for compressed words. In: *Jewels are Forever*, Springer 262–272.

Rytter, W. (2004). Grammar Compression, LZ-encodings, and string algorithms with implicit input. In Díaz, J., et al. (eds.), *Proceedings of the ICALP 2004*, Lecture Notes in Computer Science, vol. 3142, Springer-Verlag 15–27.

Salzer, G. (1992). The unification of infinite sets of terms and its applications. In: *Proceedings of the LPAR '92*, Lecture Notes in Computer Science, vol. 624, 409–420.

Schmidt-Schauß, M. (2005). Polynomial equality testing for terms with shared substructures. Frank report 21, Institut für Informatik. FB Informatik und Mathematik. Goethe-Universität Frankfurt.

Schmidt-Schauß, M. (2012). Matching of compressed patterns with character-variables. In: Tiwari, A., (ed.) *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications (RTA '12)*, Leibniz International Proceedings in Informatics, vol. 624, Schloss Dagstuhl 272–287.

Schmidt-Schauß, M., Sabel, D. and Anis, A. (2011). Congruence closure of compressed terms in polynomial time. In: *Proceedings of the FroCos*, Lecture Notes in Computer, vol. 624, Springer 227–242.

Schmidt-Schauß, M. and Schnitger, G. (2012). Fast equality test for straight-line compressed strings. *Information Processing Letters* **112** (8–9) 341–345.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* **23** (3) 337–343.

https://doi.org/10.1017/S0960129518000208 Published online by Cambridge University Press