Generalized Straight-Line Programs

Gonzalo Navarro[1,2*], Francisco Olivares[1,2*] and Cristian Urbina[1,2*]

[1]Department of Computer Science, University of Chile, Chile.
[2]CeBiB — Centre for Biotechnology and Bioengineering, Chile.

*Corresponding author(s). E-mail(s): crurbina@dcc.uchile.cl; gnavarro@uchile.cl; folivares@uchile.cl;

Abstract

It was recently proved that any Straight-Line Program (SLP) generating a given string can be transformed in linear time into an equivalent balanced SLP of the same asymptotic size. We generalize this proof to a general class of grammars we call Generalized SLPs (GSLPs), which allow rules of the form $A \rightarrow x$ where x is any Turing-complete representation (of size $|x|$) of a sequence of symbols (potentially much longer than $|x|$). We then specialize GSLPs to so-called Iterated SLPs (ISLPs), which allow rules of the form $A \rightarrow \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ of size $\mathcal{O}(t)$. We prove that ISLPs break, for some text families, the measure δ based on substring complexity, a lower bound for most measures and compressors exploiting repetitiveness. Further, ISLPs can extract any substring of length λ, from the represented text $T[1\,..\,n]$, in time $\mathcal{O}(\lambda + \log^2 n \log\log n)$. This is the first compressed representation for repetitive texts breaking δ while, at the same time, supporting direct access to arbitrary text symbols in polylogarithmic time. We also show how to compute some substring queries, like range minima and next/previous smaller value, in time $\mathcal{O}(\log^2 n \log\log n)$. Finally, we further specialize the grammars to Run-Length SLPs (RLSLPs), which restrict the rules allowed by ISLPs to the form $A \rightarrow B^t$. Apart from inheriting all the previous results with the term $\log^2 n \log\log n$ reduced to the near-optimal $\log n$, we show that RLSLPs can exploit balancedness to efficiently compute a wide class of substring queries we call "composable"—i.e., $f(X \cdot Y)$ can be obtained from $f(X)$ and $f(Y)$. As an example, we show how to compute Karp-Rabin fingerprints of texts substrings in $\mathcal{O}(\log n)$ time. While the results on RLSLPs were already known, ours are much simpler and require little precomputation time and extra data associated with the grammar.

Keywords: Grammar compression, Substring complexity, Repetitiveness measures, Substring queries

1 Introduction

Motivated by the data deluge, and by the observed phenomenon that many of the fastest-growing text collections are highly repetitive, recent years have witnessed an increasing interest in (1) defining measures of compressibility that are useful for highly repetitive texts, (2) developing compressed text representations whose size can be bounded in terms of those measures, and (3) providing efficient (i.e., polylogarithmic time) access methods to those compressed texts, so that algorithms can be run on them without ever decompressing the texts [35, 36]. We call *lower-bounding measures* those satisfying (1), *reachable measures* those (asymptotically) reached by the size of a compressed representation (2), and *accessible measures* those reached by the size of representations satisfying (3).

For example, the size γ of the smallest "string attractor" of a text T is a lower-bounding measure, unknown to be reachable [25], and smaller than the size reached by known compressors. The size b of the smallest "bidirectional macro scheme" of T [45], and the size z of the Lempel-Ziv parse of T [31], are reachable measures. The size g of the smallest context-free grammar generating (only) T [9] is an accessible measure [5]. It holds $\gamma \leq b \leq z \leq g$ for every text.

One of the most attractive lower-bounding measures devised so far is δ [10, 43]. Let $T[1\,..\,n]$ be a text over alphabet $[1\,..\,\sigma]$, and T_k be the number of distinct substrings of length k in T, which define its so-called substring complexity. Then the measure is $\delta(T) = \max_k T_k/k$. This measure has several attractive properties: it can be computed in linear time and lower-bounds all previous measures of compressibility, including γ, for every text. While δ is known to be unreachable, the measure $\delta' = \delta \log \frac{n \log \sigma}{\delta \log n}$ has all the desired properties: $\Omega(\delta')$ is the space needed to represent some text family for each n, σ, and δ; within $\mathcal{O}(\delta')$ space it is possible to represent every text T and access any length-λ substring of T in time $\mathcal{O}(\lambda + \log n)$ [28], together with more powerful operations [24, 28, 29].

As for g, a *straight-line program (SLP)* is a context-free grammar that generates (only) T, and has size-2 rules of the form $A \rightarrow BC$, where B and C are nonterminals, and size-1 rules $A \rightarrow a$, where a is a terminal symbol. The SLP size is the sum of all its rule sizes. A *run-length SLP (RLSLP)* may contain, in addition, size-2 rules of the form $A \rightarrow B^t$, representing t repetitions of nonterminal B. An RLSLP of size g_{rl} can be represented in $\mathcal{O}(g_{rl})$ space, and within that space we can offer fast string access and other operations [10, App. A]. It holds $\delta \leq g_{rl} = \mathcal{O}(\delta')$, where g_{rl} is the smallest RLSLP that generates T [28, 35] (the size g of the smallest grammar or SLP, instead, is not always $\mathcal{O}(\delta')$).

While δ lower-bounds all previous measures on every text, δ' is not the smallest accessible measure. In particular, g_{rl} is always $\mathcal{O}(\delta')$, and it can be smaller by up to a logarithmic factor. Indeed, g_{rl} is a minimal accessible measure as far as we know (but see our Conclusions). It is asymptotically between z and g [35]. An incomparable accessible measure is $z_{end} \geq z$, the size of the LZ-End parse of the text [26, 30].

The recently proposed L-systems [37, 38] show, in turn, that δ is not a lower bound to every reachable measure. L-systems are like SLPs where all the symbols are nonterminals and the derivation ends at a specified depth in the derivation tree.

The size ℓ of the smallest L-system generating $T[1..n]$ is a reachable measure of repetitiveness and was shown to be as small as $\mathcal{O}(\delta/\sqrt{n})$ on some text families, thereby sharply breaking δ as a lower bound. Measure ℓ, however, is unknown to be accessible, and thus one may wonder whether there exist accessible text representations that are smaller than δ.

In this paper we present several contributions to this state of the art. All our time complexities hold under the transdichotomous RAM model of computation, with word size $w = \Theta(\log n)$ bits. Our space is also measured in w-bit words.

1. We extend the result of Ganardi et al. [17], which shows that any SLP of size g generating a text of length n can be balanced to produce another SLP of size $\mathcal{O}(g)$ whose derivation tree is of height $\mathcal{O}(\log n)$. Our extension is called *Generalized SLPs (GSLPs)*, which allow rules of the form $A \to x$ (of size $|x|$), where x is a *program* (in any Turing-complete formalism) that outputs the right-hand side of the rule. We show that, if every nonterminal appearing in x's output occurs at least twice, then the GSLP can be balanced in the same way as SLPs.
2. We explore a particular case of GSLP we call *Iterated SLPs (ISLPs)*. ISLPs extend SLPs (and RLSLPs) by allowing a more complex version of the rule $A \to B^t$, namely $A \to \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, of size $2+2t$. We show that some text families are generated by an ISLP of size $\mathcal{O}(\delta/\sqrt{n})$, thereby sharply breaking the $\Omega(\delta)$ barrier.
3. Using the fact that ISLPs are GSLPs and thus can be balanced, we show how to extract a substring of length λ from the ISLP in time $\mathcal{O}(\lambda + \log^2 n \log\log n)$, as well as computing substring queries like range minimum and next/previous smaller value, in time $\mathcal{O}(\log^2 n \log\log n)$. ISLPs are thus the first accessible representation that can reach size $o(\delta)$ on some text families..
4. Finally, we apply the balancing result to RLSLPs, which allow rules of the form $A \to B^t$. While the results on ISLPs are directly inherited (because RLSLPs are ISLPs) with the polylogs becoming just the nearly-optimal $\mathcal{O}(\log n)$ [46], we give a general technique to compute a wide family of "composable" queries f on substrings (i.e., $f(X \cdot Y)$ can be computed from $f(X)$ and $f(Y)$). As an application, we show how to compute Karp-Rabin fingerprints on text substrings in time $\mathcal{O}(\log n)$, which we do not know how to do efficiently on ISLPs. This considerably simplifies and extends previous results [10, App. A], as balanced grammars enable simpler algorithms that do not require large and complex additional structures.

This work is an extended version of articles published in SPIRE 2022 [41] and LATIN 2024 [39], which are now integrated into a coherent framework in which specialized results are derived from more general ones, new operations are supported, and proofs are complete.

2 Preliminaries

We explain some concepts and notation used in the rest of the paper.

2.1 Strings

Let $\Sigma = [1\,..\,\sigma]$ be an ordered set of symbols called the *alphabet*. A *string* $T[1\,..\,n]$ of *length* n is a finite sequence $T[1]\,T[2]\cdots T[n]$ of n symbols in Σ. We denote by ε the unique string of length 0. We denote by Σ^* the set of all finite strings with symbols in Σ. The i-th symbol of T is denoted by $T[i]$; the notation $T[i\,..\,j]$ stands for the sequence $T[i]\cdots T[j]$ for $1 \leq i \leq j \leq n$, and ε otherwise. The *concatenation* of $X[1\,..\,n]$ and $Y[1\,..\,m]$ is defined as $X \cdot Y = X[1]\cdots X[n]\,Y[1]\cdots Y[m]$ (we omit the dot when there is no ambiguity). If $T = XYZ$, then X (resp. Y, resp. Z) is a *prefix* (resp. *substring*, resp. *suffix*) of T. A prefix (resp. suffix) X of T is *non-empty* if $X \neq \varepsilon$, *proper* if $X \neq T$, and *non-trivial* if it is both non-empty and proper. A *power* T^k stands for k consecutive concatenations of the string T. We denote by $|T|_a$ the number of occurrences of the symbol a in T. A *string morphism* is a function $\varphi : \Sigma^* \to \Sigma^*$ such that $\varphi(xy) = \varphi(x) \cdot \varphi(y)$ for any strings x and y.

2.2 Straight-Line Programs

A *straight-line program* (SLP) is a context-free grammar [44] that contains exactly one rule per nonterminal A, which can only be a terminal rule $A \to a$ for some $a \in \Sigma$, or a binary rule $A \to BC$ for variables B and C whose derivations cannot reach again A. These restrictions ensure that each variable of the SLP generates a unique string, defined as $\texttt{exp}(A) = a$ for a rule $A \to a$, and as $\texttt{exp}(A) = \texttt{exp}(B) \cdot \texttt{exp}(C)$ for a rule $A \to BC$. A *run-length straight-line program* (RLSLP) is an SLP that also admits run-length rules of the form $A \to B^t$ for some $t \geq 3$, with their expansion defined as $\texttt{exp}(A) = \texttt{exp}(B)^t$. The *size* of an SLP G, denoted $|G|$, is the sum of the lengths of the right-hand sides of its rules. The size of an RLSLP is defined similary, assuming that rules $A \to B^t$ are of size 2 (i.e., two integers to represent B and t). We use $\texttt{exp}(G)$ to refer to $\texttt{exp}(S)$, where S is the initial rule of G, thus $\texttt{exp}(G) = T$.

The *derivation or parse tree* of an SLP is an ordinal tree where the nodes are the variables, the root is the initial variable, and the leaves are the terminal variables. The children of a node are the variables appearing in the right-hand side of its rule (in left-to-right order). The *height* of an SLP or RLSLP is the length of the longest path from the root to a leaf node in its derivation tree. The derivation tree of RLSLPs is analogous to that of SLPs; the nodes labeled A, for the rules $A \to B^t$, have t children labeled B.

The *grammar tree* is obtained by pruning the parse tree so that only the leftmost occurrence of a nonterminal is retained as an internal node and all the others become leaves. Rules $A \to B^t$ are represented as the node A having a left child B (which can be internal or a leaf) and a special right child denoting B^{t-1} (which is a leaf). It is easy to see that the grammar tree has exactly $|G| + 1$ nodes.

SLPs and RLSLPs yield measures of repetitiveness g and g_{rl}, defined as the size of the smallest SLP and RLSLP generating the text, respectively. Clearly, it holds that $g_{rl} \leq g$. It also has been proven that both g and g_{rl} are NP-hard to compute [9, 22].

2.3 Other Repetitiveness Measures

For self-containedness, we describe the most important repetitiveness measures and relate them with the accessible measures g and g_{rl}; for more details see a survey [35].

Burrows-Wheeler Transform.

The *Burrows-Wheeler Transform* (BWT) [8] is a reversible rearrangement of the symbols of T, which we denote by $\mathtt{bwt}(T)$. It is obtained by sorting lexicographically all the rotations of the string T and concatenating their last symbols, which can be done in $\mathcal{O}(n)$ time. The measure r is defined as the size of the *run-length encoding* of $\mathtt{bwt}(T)$. Usually, T is assumed to be appended with a sentinel symbol \$ strictly smaller than any other symbol in T, and then we call $r_{\$}$ the size of the run-length encoding of $\mathtt{bwt}(T\$)$. This measure is then reachable, and fully-functional indexes of size $\mathcal{O}(r_{\$})$ exist [15], but interestingly, it is unknown to be accessible. While this measure is generally larger than others, it can be upper-bounded by $r_{\$} = \mathcal{O}(\delta \log \delta \log \frac{n}{\delta})$ [23].

Lempel-Ziv Parsing.

The *Lempel-Ziv parsing* (LZ) [31] of a text $T[1\,..\,n]$ is a *factorization* into non-empty *phrases* $T = X_1 X_2 \cdots X_z$ where each X_i is either the first occurrence of a symbol or the longest prefix of $X_i \cdots X_z$ with a copy in T starting at a position in $[1\,..\,|X_1 \cdots X_{i-1}|]$. LZ is called a *left-to-right* parsing because each phrase has its *source* starting to the left, and it is optimal (i.e., with the smallest z) among all parsings satisfying this condition. It can be constructed greedily from left to right in $\mathcal{O}(n)$ time. The measure z is defined as the number of phrases in the LZ parsing of the text, and it has been proved that $z \le g_{rl}$ [40]. While z is obviously reachable, it is unknown to be accessible. A close variant $z_{end} \ge z$ [30] that forces phrase sources to be end-aligned with a preceding phrase, has been shown to be accessible [26].

Bidirectional Macro Schemes.

A *bidirectional macro scheme* (BMS) [45] is a factorization of a text $T[1\,..\,n]$ where each phrase can have its source starting either to the left or to the right. The only requeriment is that by transitively following the pointers from phrases to sources, we always reach an explicit symbol, that is, we do not fall in loops. The measure b is defined as the size of the smallest BMS representing the text. Clearly, b is reachable, but it is unknown to be accessible. It holds that $b \le z$, and it was proved that $b \le 2r_{\$}$ [40]. Computing b is NP-hard [16].

String Attractors.

A *string attractor* for a text $T[1\,..\,n]$ is a set of positions $\Gamma \subseteq [1\,..\,n]$ such that any substring of $T[i\,..\,j]$ has an occurrence $T[i'\,..\,j']$ crossing at least one of the positions in Γ (i.e., there exist $k \in \Gamma$ such that $i' \le k \le j'$). The measure γ is defined as the size of the smallest string attractor for the string T, and it is NP-hard to compute [25]. It holds that string attractors asymptotically lower bound bidirectional macro schemes, that is, $\gamma = \mathcal{O}(b)$, and can sometimes be asymptotically strictly smaller [3]. On the other hand, it is unknown whether γ is reachable.

Substring Complexity.

Let $T[1..n]$ be a text and T_k be the number of distinct substrings of length k in T, which define its so-called substring complexity. Then the measure is $\delta = \max_k T_k/k$ [10, 43]. This measure can be computed in $\mathcal{O}(n)$ time and lower-bounds γ, and thus all previous measures of compressibility, for every text. On the other hand, it is known to be unreachable [28]. The related measure $\delta' = \delta \log \frac{n \log \sigma}{\delta \log n}$ is reachable and accessible, and is asymptotically optimal on some text family for every n, σ, and δ [28]. Besides, g_{rl} (and thus z, b, and γ, but not g) are upper-bounded by $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$; g is upper-bounded by $\mathcal{O}(\gamma \log^2 \frac{n}{\gamma})$ [25, 28].

L-systems.

An *L-system* (for compression) is a tuple $L = (V, \varphi, \tau, S, d, n)$ extending a traditional Lindenmayer system [32, 33], where V is the set of variables (which are also considered as terminal symbols), $\varphi : V \to V^+$ is the set of rules (and also a morphism of strings), $\tau : V \to V$ is a coding, $S \in V$ the initial variable, and d and n are integers. The string generated by the system is $\tau(\varphi^d(S))[1..n]$. The measure ℓ is defined as the size of the smallest L-system generating the string. It has been proven that ℓ is incomparable to δ (ℓ can be smaller by a $\sqrt{n}$ factor) and almost any other repetitiveness measure considered in the literature [37, 38].

3 Generalized SLPs and How to Balance Them

We introduce a new class of SLP which we show can be balanced so that its derivation tree is of height $\mathcal{O}(\log n)$.

Definition 1. *A* generalized straight-line program *(GSLP) is an SLP that allows special rules of the form $A \to x$, where x is a* program *(in any Turing-complete language) of length $|x|$ whose output* $\texttt{OUT}(x)$ *is a nonempty sequence of variables, none of which can reach A. The rule $A \to x$ contributes $|x|$ to the size of the GSLP; the standard SLP rules contribute as usual. A special rule $A \to x$ is said to be* balanceable *if every variable occurring in* $\texttt{OUT}(x)$ *appears at least twice on it. A GSLP is said to be* balanceable *if all its special rules are balanceable.*

We can choose any desired language to describe the programs x. Though in principle $|x|$ can be taken as the Kolmogorov complexity of $\texttt{OUT}(x)$, we will focus on very simple programs and on the asymptotic value of $|x|$.

We will prove that any balanceable GSLP can be balanced without increasing its asymptotic size. Our proof generalizes that of Ganardi et al. [17, Thm. 1.2] for SLPs in a similar way to how it was extended to balance RLSLPs [41]. Just as Ganardi et al., in this section we will allow SLPs to have rules of the form $A \to B_1 \cdots B_t$, of size t, where each B_j is a terminal or a nonterminal; this can be converted into a strict SLP of the same asymptotic size.

We introduce some definitions and state some results, from the work of Ganardi et al. [17], that we need in order to prove our balancing result for GSLPs.

A *directed acyclic graph* (DAG) is a directed multigraph $D = (V, E)$ without cycles (nor loops). We denote by $|D|$ the number of edges in this DAG. For our purposes, we assume that any DAG has a distinguished node r called the *root*, satisfying that any other node can be reached from r and r has no incoming edges. We also assume that if a node has k outgoing edges, they are numbered from 1 to k, so edges are of the form (u, i, v). The *sink nodes* of a DAG are the nodes without outgoing edges. The set of sink nodes of D is denoted by W. We denote the number of paths from u to v as $\pi(u, v)$, and $\pi(u, V) = \sum_{v \in V} \pi(u, v)$ for a set V of nodes. The number of paths from the root to the sink nodes is $n(D) = \pi(r, W)$.

One can interpret an SLP G generating a string T as a DAG D: There is a node for each variable in the SLP, the root node is the initial variable, variables of the form $A \to a$ are the sink nodes, and a variable with rule $A \to B_1 B_2 \cdots B_t$ has outgoing edges (A, i, B_i) for $i \in [1..t]$. Note that if D is a DAG representing G, then $n(D) = |\mathtt{exp}(G)| = |T|$.

Definition 2. *(Ganardi et al. [17, page 5]) Let $D = (V, E)$ be a DAG, and define the pairs $\lambda(v) = (\lfloor \log_2 \pi(r, v) \rfloor, \lfloor \log_2 \pi(v, W) \rfloor)$ for every $v \in V$. The* symmetric centroid decomposition (SC-decomposition) *of a DAG D produces a set of edges between connected nodes with the same λ pairs defined as $E_{scd}(D) = \{(u, i, v) \in E \mid \lambda(u) = \lambda(v)\}$, partitioning D into disjoint paths called* SC-paths *(some of them possibly of length 0).*

The set E_{scd} can be computed in $\mathcal{O}(|D|)$ time. If D is the DAG of an SLP G, then $|D|$ is $O(|G|)$. The following lemma justifies the name "SC-paths".

Lemma 1. *(Ganardi et al. [17, Lemma 2.1]) Let $D = (V, E)$ be a DAG. Then every node has at most one outgoing and at most one incoming edge from $E_{scd}(D)$. Furthermore, every path from the root r to a sink node contains at most $2 \log_2 n(D)$ edges that do not belong to $E_{scd}(D)$.*

Note that the sum of the lengths of all SC-paths is at most the number of nodes of the DAG, or equivalently, the number of variables of the SLP.

The following definition and technical lemma are needed to construct the building blocks of our balanced GSLPs.

Definition 3. *(Ganardi et al. [17, page 7]) A* weighted string *is a string $T \in \Sigma^*$ equipped with a* weight function $||\cdot|| : \Sigma \to \mathbb{N}\backslash\{0\}$, *which is extended homomorphically. If A is a variable in an SLP G, then we write $||A||$ for the weight of the string* $\mathtt{exp}(A)$ *derived from A.*

Lemma 2. *(Ganardi et al. [17, Proposition 2.2]) For every non-empty weighted string T of length n one can construct in linear time an SLP G generating T with the following properties:*

- *G contains at most $3n$ variables.*
- *All right-hand sides of G have length at most 4.*

- *G contains suffix variables*[1] $S_1, \ldots, S_n$ *producing all non-empty suffixes of* T.
- *every path from* S_i *to some terminal symbol* a *in the derivation tree of* G *has length at most* $3 + 2(\log_2 ||S_i|| - \log_2 ||a||)$.

With this machinery, we are ready to prove the main result of this section. Note that we require that GSLP's special rules are always the endpoint of some SC-path, which makes the argument of Ganardi et al. [17] for regular SLPs easily applicable to GSLPs: the balancing procedure does not involve the special variables of the GSLPs.

Theorem 3. *Given a balanceable GSLP G generating a string T, it is possible to construct an equivalent GSLP G' of size $\mathcal{O}(|G|)$ and height $\mathcal{O}(\log n)$ in $\mathcal{O}(|G| + t(G))$ time, where $t(G)$ is the time needed to compute the lengths of the expansion of each variable in G.*

Proof. Transform the GSLP G into an SLP H by (conceptually) replacing their special rules $A \to x$ by $A \to \mathtt{OUT}(x)$, and then obtain the SC-decomposition $E_{scd}(D)$ of the DAG D of H. Observe that the SC-paths of H use the same variables of G, so it holds that the sum of the lengths of all the SC-paths of H is less than the number of variables of G. Also, note that any special variable $A \to x$ of G is necessarily the endpoint (i.e., the last node of a directed path) of an SC-path in D. To see this note that $\lambda(A) \neq \lambda(B)$ for any B that appears in $\mathtt{OUT}(x)$, because $\log_2 \pi(A, W) \geq \log_2(|\mathtt{OUT}(x)|_B \cdot \pi(B, W)) \geq 1 + \log_2 \pi(B, W)$, where $|\mathtt{OUT}(x)|_B$ is the number of occurrences of B within $\mathtt{OUT}(x)$—so $|\mathtt{OUT}(x)|_B \geq 2$ because G is balanceable. This implies that the balancing procedure of Ganardi et al. on H, which transforms the rules of variables that are not the endpoint of an SC-path in the DAG D, will not touch variables that were originally special variables in G.

Let $\rho = (A_0, d_0, A_1), (A_1, d_1, A_2), \ldots, (A_{p-1}, d_{p-1}, A_p)$ be an SC-path of D. It holds that for each A_i with $i \in [0\,..\,p-1]$, in the SLP H its rule goes to two distinct variables, one to the left and one to the right. Thus, for each variable A_i, with $i \in [0\,..\,p-1]$, there is a variable A'_{i+1} that is not part of the path. Let $A'_1 A'_2 \cdots A'_p$ be the sequence of these variables. Let $L = L_1 L_2 \cdots L_s$ be the subsequence of left variables of the previous sequence. Then construct an SLP of size $\mathcal{O}(s) \subseteq \mathcal{O}(p)$ for the sequence L (seen as a string) as in Lemma 2, using $|\mathtt{exp}(L_i)|$ in H as the weight function. In this SLP, any path from the suffix nonterminal S_i to a variable L_j has length at most $3 + 2(\log_2 ||S_i|| - \log_2 ||L_j||)$. Similarly, construct an SLP of size $\mathcal{O}(t) \subseteq \mathcal{O}(p)$ for the sequence $R = R_1 R_2 \cdots R_t$ of right symbols in reverse order, as in Lemma 2, but with prefix variables P_i instead of suffix variables. Each variable A_i, with $i \in [0\,..\,p-1]$, derives the same string as $w_l A_p w_r$, for some suffix w_l of L and some prefix w_r of R. We can find rules deriving these prefixes and suffixes in the SLPs produced in the previous step, so for any variable A_i, we construct an equivalent rule of length at most 3. Add these equivalent rules, and the left and right SLP rules to a new GSLP G'. Do this for all SC-paths. Finally, add the original terminal variables and special variables (which are left unmodified) of the GSLP G, so G' is a GSLP equivalent to G.

Figure 1 shows an example where the special GSLP rules are of the form $A \to B^t$, meaning t copies of B (i.e., the GSLP is an RLSLP).

[1] Namely, $\mathtt{exp}(S_i) = T[i..n]$, for $i \in [1..n]$.

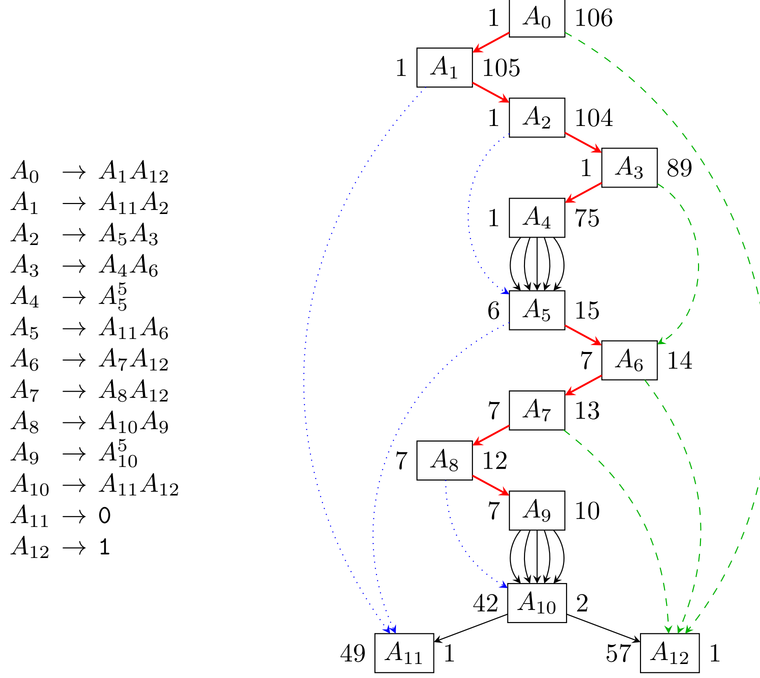

Fig. 1 The DAG and SC-decomposition of an unfolded RLSLP generating the string $\mathtt{0(0(01)^6 1^2)^6 (01)^5 1^3}$. The value to the left of a node is the number of paths from the root to that node, and the value to the right is the number of paths from the node to sink nodes. Thick red edges belong to the SC-decomposition of the DAG. Dotted blue (resp. dashed green) edges branch from an SC-path to the left (resp. to the right).

The SLP constructed for L has all its rules of length at most 4, and $3s \leq 3p$ variables. The same happens with R. The other constructed rules also have a length of at most 3, and there are p of them. Summing over all SC-paths, we have $\mathcal{O}(|G|)$ size. The special variables cannot sum up to more than $\mathcal{O}(|G|)$ size. Thus, the GSLP G' has size $\mathcal{O}(|G|)$.

Any path in the derivation tree of G' is of length $\mathcal{O}(\log n)$. To see why, let $A_0, \ldots, A_p$ be an SC-path. Consider a path from a variable A_i with $i \in [0, p]$, to the occurrence of some variable within the string of variables produced by the right-hand side of A_p in G'. Clearly, this path has length at most 2 (i.e., from A_i to A_p and from A_p to such variable). Now consider a path from A_i to a variable A'_j in the sequence L of left variables of the SC-path, with $i < j \leq p$ (that is, A'_j is a variable that diverges from the SC-path before reaching A_p). By construction this path is of the form $A_i \to S_k \to^* A'_j$ (where $\to^*$ represents a path) for some suffix variable S_k (if the occurrence of A'_j is a left symbol), and per Lemma 2 its length is at most $1 + 3 + 2(\log_2 ||S_k|| - \log_2 ||A'_j||) \leq 4 + 2\log_2 ||A_i|| - 2\log_2 ||A'_j||$. Analogously, if A'_j is a right variable, the length of the path is bounded by $1 + 3 + 2(\log_2 ||P_k|| - \log_2 ||A'_j||) \leq 4 + 2\log_2 ||A_i|| - 2\log_2 ||A'_j||$ (where P_k is some prefix variable). We call all these paths whose length we bounded *weight-balanced paths*.

Now consider a maximal path from the root to a leaf in the derivation tree of G'. Factorize it as

$$A_0 \to^* A_1 \to^* \cdots \to^* A_k$$

where each A_i is a variable of H (and also of G and D), and in between each A_i and A_{i+1}, in the DAG D there is almost an SC-path, except that the last edge is not in E_{scd}. Each path $A_i \to^* A_{i+1}$ is a weight-balanced path in the constructed GSLP G'. Simply put, in G', either the path goes directly from A_i to the SC-path endpoint and follows an edge from there to A_{i+1}, or it goes through a suffix (or prefix) variable. In either case, the length of these paths is bounded by 2 or by $4 + 2\log_2 ||A_i|| - 2\log_2 ||A_{i+1}||$), respectively.

The length of the full path from root to leaf in G' is then at most

$$\sum_{i=0}^{k-1}(4 + 2\log_2 ||A_i|| - 2\log_2 ||A_{i+1}||) \leq 4k + 2\log_2 ||A_0|| - 2\log_2 ||A_k||$$

By Lemma 1, $k \leq 2\log_2 n$, which yields the upper bound $\mathcal{O}(\log n)$.

To have standard SLP rules of size at most two, delete rules in G' of the form $A \to B$ (replacing all A's by B's), and note that rules of the form $A \to BCDE$ or $A \to BCD$ can be decomposed into rules of length 2, with only a constant increase in size and depth.

The balancing procedure uses $\mathcal{O}(|G| + t(G))$ time and $\mathcal{O}(|G| + s(G))$ auxiliary space, where $t(G)$ and $s(G)$ are the time and space needed to compute and store the set of all the pairs $(B, |\texttt{OUT}(x)|_B)$, where B appears $|\texttt{OUT}(x)|_B > 0$ times in $\texttt{OUT}(x)$, for every special variable $A \to x$. With this information the set $E_{scd}(D)$ can be computed in $\mathcal{O}(|G| + t(G))$ time, instead of $\mathcal{O}(|H|)$ time. The SLPs of Lemma 2 are constructed in linear time in the lengths of the SC-paths, which sum to $\mathcal{O}(|G|)$ in total. □

4 Iterated Straight-Line Programs

We now define iterated SLPs and show that they can be much smaller than δ.

Definition 4. *An* iterated straight-line program *of* degree d *(d-ISLP) is an SLP that allows in addition* iteration rules *of the form*

$$A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$$

where $1 \leq k_1, k_2$, $0 \leq c_1, \ldots, c_t \leq d$ *are integers,* $B_1, \ldots, B_t$ *are variables that cannot reach* A *(so the ISLP generates a unique string), and the product of strings refers to their concatenation. Iteration rules have size* $2 + 2t = \mathcal{O}(t)$ *and expand to*

$$\texttt{exp}(A) = \prod_{i=k_1}^{k_2} \texttt{exp}(B_1)^{i^{c_1}} \cdots \texttt{exp}(B_t)^{i^{c_t}}$$

where if $k_1 > k_2$ the iteration goes from $i = k_1$ downwards to $i = k_2$. The size $size(G)$ of a d-ISLP G is the sum of the sizes of all of its rules.

Definition 5. *The measure $g_{it(d)}(T)$ is defined as the size of the smallest d-ISLP that generates T, whereas $g_{it}(T) = \min_{d\geq 0} g_{it(d)}(T)$.*

The following observations show that ISLPs subsume RLSLPs, and thus, can be smaller than the smallest L-system.

Proposition 4. *For any $d \geq 0$, it always holds that $g_{it(d)} = \mathcal{O}(g_{rl})$.*

Proof. Just note that a rule $A \to \prod_{i=1}^{t} B^{i^0}$ from an ISLP simulates a rule $A \to B^t$ from a RLSLP. In particular, 0-ISLPs are equivalent to RLSLPs. $\square$

Proposition 5. *For any $d \geq 0$, there exists a string family where $g_{it(d)} = o(\ell)$.*

Proof. Navarro and Urbina show a string family where $g_{rl} = o(\ell)$ [38]. Hence, $g_{it(d)}$ is also $o(\ell)$ in this family. $\square$

We now show that $d = 1$ suffices to obtain ISLPs that are significantly smaller than δ for some string families.

Lemma 6. *Let $d \geq 1$. There exists a string family with $g_{it(d)} = \mathcal{O}(1)$ and $\delta = \Omega(\sqrt{n})$.*

Proof. Such a family is formed by the strings $s_k = \prod_{i=1}^{k} \mathtt{a}^i\mathtt{b}$. The 1-ISLPs with initial rule $S_k \to \prod_{i=1}^{k} A^i B$, and rules $A \to \mathtt{a}$, $B \to \mathtt{b}$, generate each string s_k in the family using $\mathcal{O}(1)$ space. On the other hand, it holds that $\delta = \Omega(\sqrt{n})$ in the family $\mathtt{c}s_k$ [38]. As δ can only decrease by 1 after the deletion of a character [1], $\delta = \Omega(\sqrt{n})$ in the family s_k too. $\square$

On the other hand, ISLPs can perform worse than other compressed representations; recall that $\delta \leq \gamma \leq b \leq r_\$$.

Lemma 7. *Let $\mu \in \{r, r_\$, \ell\}$. For any $d \geq 0$, there exists a string family with $g_{it(d)} = \Omega(\log n)$ and $\mu = \mathcal{O}(1)$.*

Proof. Consider the family of Fibonacci words defined recursively as $F_0 = \mathtt{a}$, $F_1 = \mathtt{b}$, and $F_{i+2} = F_{i+1}F_i$ for $i \geq 0$. Fibonacci words cannot contain substrings of the form x^4 for any $x \neq \varepsilon$ [20]. Consider an ISLP for a Fibonacci word and a rule of the form $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$. Observe that if $c_r \neq 0$ for some r, then $\max(k_1, k_2) < 4$, as otherwise $\mathtt{exp}(B_r)^4$ occurs in T. Similarly, if $c_r = 0$ for all r, then $|k_1 - k_2| < 3$, as otherwise $\mathtt{exp}(B_1 \cdots B_t)^4$ appears in T. In the latter case, we can rewrite the product with $k_1, k_2 \in [1..3]$. Therefore, we can unfold the product rule into standard SLP rules of total size at most $9t$ ($3t$ variables raised to at most 3 each because we assumed our word is Fibonacci). Hence, for any d-ISLP G generating a Fibonacci word, there is an SLP G' of size $\mathcal{O}(|G|)$ generating the same string. As $g = \Omega(\log n)$ in every string family [35], we obtain that $g_{it(d)} = \Omega(\log n)$ in this family too. On the other hand, $r_\$, r$, and ℓ are $\mathcal{O}(1)$ in the even Fibonacci words [34, 37, 40]. $\square$

Lemma 8. *For any $d \geq 0$, there exists a string family satisfying that $z = \mathcal{O}(\log n)$ and $g_{it(d)} = \Omega(\log^2 n / \log\log n)$.*

Proof. Let $T(n)$ be the length n prefix of the infinite Thue-Morse word[2] [2] on the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Let $k_1, ..., k_p$ be a set of distinct positive integers, and consider strings of the form $S = T(k_1)|_1 T(k_2)|_2 \cdots T(k_{p-1})|_{p-1} T(k_p)$, where $|_i$'s are unique separators and k_1 is the largest of the k_i. Since the sequences $T(k_i)$ are cube-free[3] [2], there is no asymptotic difference in the size of the smallest SLP and the smallest ISLP (similarly to Lemma 7) for the string S. Hence, $g_{id(d)} = \Theta(g)$ in this family. It has been proved that $g = \Omega(\log^2 k_1 / \log\log k_1)$ and $z = \mathcal{O}(\log k_1)$ for some specific sets of integers where $p = \Theta(k_1)$ [7]. Thus, the result follows. ☐

One thing that makes ISLPs robust is that they are not very sensitive to reversals, morphism application, or edit operations (insertions, deletions, and substitutions of a single character). The measure $g_{it(d)}$ behaves similarly to SLPs in this matter, for which it has been proved that $g(T') \leq 2g(T)$ after an edit operation that converts T to T' [1], and that $g(\varphi(T)) \leq g(T) + c_\varphi$ with c_φ a constant depending only on the morphism φ [12]. This makes $g_{it(d)}$ much more robust to string operations than measures like r and $r_\$$, which are highly sensitive to all these transformations [1, 12, 18, 19].

Lemma 9. *Let G be a d-ISLP generating T. Then there exists a d-ISLP of size $|G|$ generating the reversed text T^R. Let φ be a morphism. Then there exists a d-ISLP of size $|G| + c_\varphi$ generating the text $\varphi(T)$, where c_φ is a constant depending only on φ. Moreover, there exists a d-ISLP of size at most $\mathcal{O}(|G|)$ generating T' where T and T' differ by one edit operation.*

Proof. For the first claim, note that reversing all the SLP rules and expressions inside the special rules, and swapping the values k_1 and k_2 in each special rule is enough to obtain a d-ISLP of the same size generating T^R.

For the second claim, we replace rules of the form $A \rightarrow \mathtt{a}$ with $A \rightarrow \varphi(\mathtt{a})$, yielding a grammar of size less than $|G| + \sum_{a \in \Sigma} |\varphi(a)|$. Then we replace these rules with binary rules, which asymptotically do not increase the size of the grammar.

For the edit operations, we proceed as follows. Consider the derivation tree of the ISLP, and the path from the root to the character we want to substitute, delete, or insert a character before or after. Then, we follow this path in a bottom up manner, constructing a new variable A' for each node A we visit. We start at some $A \rightarrow \mathtt{a}$, so we construct $A' \rightarrow x$ where either $x = \mathtt{c}$ or $x = \mathtt{ac}$ or $x = \mathtt{ca}$ or $x = \varepsilon$ depending on the edit operation. If we reach a node $A \rightarrow BC$ going up from B (so we already constructed B'), we construct a node $A' \rightarrow B'C$ (analogously if we come from C). If we reach a node $A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ going up from a specific B_r with $r \in [1..t]$ (so we already constructed B'_r) at the k-th iteration of the product with $k_1 \leq k \leq k_2$

[2] This is the binary infinite sequence obtained by staring with 0 and appending the binary complement of the string obtained so far, that is, 0 1 10 1001 10010110 . . .

[3] Those are the sequences that do not contain three consecutive identical substrings.

and being the q-th copy of B_r inside $B_r^{k^{c_r}}$, then we construct the following new rules

$$A_1 \to \prod_{i=k_1}^{k-1} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}},\ A_2 \to \prod_{i=k}^{k} B_1^{i^{c_1}} \cdots B_{r-1}^{i^{c_{r-1}}},\ A_3 \to \prod_{i=1}^{q-1} B_r^{i^0},$$
$$A_4 \to \prod_{i=q+1}^{k^{c_r}} B_r^{i^0},\ A_5 \to \prod_{i=k}^{k} B_{r+1}^{i^{c_{r+1}}} \cdots B_t^{i^{c_t}},\ A_6 \to \prod_{i=k+1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$$
$$A' \to A_1 A_2 A_3 B_r' A_4 A_5 A_6$$

which are equivalent to A (except by the modified, inserted, or deleted symbol) and sum to a total size of at most $6t+21$. As $t \geq 1$, it holds that $(6t+21)/(2t+2) \leq 7$. After finishing the whole process, we obtain a d-ISLP of size at most $8|G|$. Note that this ISLP contains ε-rules. It also contains some non-binary SLP rules, which can be transformed into binary rules, at most doubling the size of the grammar. □

5 Accessing ISLPs

We have shown that $g_{it(d)}$ breaks the lower bound δ already for $d \geq 1$. We now show that the measure is accessible. Concretely, we will prove the following result along Sections 5.2 to 5.4. Before proving it, Section 5.1 shows how the result can be specialized by properly bounding h and d. At the end, Section 5.5 extends the result to computing functions over substrings, without need of extracting them first.

Theorem 10. *Let $T[1\,..\,n]$ be generated by a d-ISLP G of height h. Then, we can build in time $\mathcal{O}((|G|+d)d\lceil d\log d/\log n\rceil)$ and space $\mathcal{O}(|G|+d\lceil d\log d/\log n\rceil)$ a data structure of size $\mathcal{O}(|G|)$ that extracts any substring of T of length λ in time $\mathcal{O}(\lambda+(h+\log n+d)d\lceil d\log d/\log n\rceil)$, using $\mathcal{O}(h+d\lceil d\log d/\log n\rceil)$ additional words of working space.*

Algorithm 1 displays our general access strategy, still without all the details. In broad terms, we extend the standard algorithm that descends along the parse tree towards $\texttt{exp}(A)[l]$. Lines 2–8 handle the classic cases, whereas lines 9–14 solve the case of our new extended rules. In this case, we first need to determine which is the value of $i \in [k_1\,..\,k_2]$ where $\texttt{exp}(A)[l]$ falls (line 10), and adjust l by subtracting the lengths of the preceding "blocks" (line 11). Second, we must determine where $\texttt{exp}(A)[l]$ falls within the ith block (line 12), and adjust l again by subtracting the length of the preceding runs of B_1 to B_{r-1} (line 13). Once we know that $\texttt{exp}(A)[l]$ falls within B_r, we easily compute the desired symbol of $\texttt{exp}(B_r)$ by which the recursion should continue, in line 14. Sections 5.2 and 5.3 will focus on how to efficiently compute the lengths of rules and block prefixes, and how to efficiently find where $\texttt{exp}(A)[l]$ falls. Some notation and useful facts will already be introduced in Section 5.1.

Algorithm 1 Direct access on d-ISLPs (simplified version)

Input: A variable A of an ISLP, and a position $l \in [1, |\texttt{exp}(A)|]$.
Output: The character $\texttt{exp}(A)[l]$.

1: **function** $\text{ACCESS}(A, l)$
2: **if** $A \to a$ **then** // found the leaf in the parse tree for $T[l]$
3: **return** a
4: **else if** $A \to BC$ **then** // go left or right as on classic SLPs
5: **if** $l \leq |\texttt{exp}(B)|$ **then**
6: **return** $\text{ACCESS}(B, l)$
7: **else** $(l > |\texttt{exp}(B)|)$
8: **return** $\text{ACCESS}(C, l - |\texttt{exp}(B)|)$
9: **else** $(A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}})$ // find the proper descendant node
10: find the value of i where $\texttt{exp}(A)[l]$ belongs
11: adjust $l \leftarrow l - |\Pi_{j=k_1}^{i-1} B_1^{j^{c_1}} \cdots B_t^{j^{c_t}}|$
12: find the run $B_r^{i^{c_r}}$ where $\texttt{exp}(A)[l]$ belongs inside $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$
13: adjust again $l \leftarrow l - |B_1^{i^{c_1}} \cdots B_{r-1}^{i^{c_{r-1}}}|$
14: **return** $\text{ACCESS}(B_r, (l - 1 \bmod |\texttt{exp}(B_r)|) + 1)$

5.1 Specializing the Result

Before proving Theorem 10, we obtain a useful special case by showing that both h and d can be bounded to $\mathcal{O}(\log n)$ without increasing the asymptotic size of G. Theorem 10 then implies the following result.

Theorem 11. *Let $T[1..n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G| + \log n)\log n \log\log n)$ and space $\mathcal{O}(|G| + \log n \log\log n)$ a data structure of size $\mathcal{O}(|G|)$ that extracts any substring of T of length λ in time $\mathcal{O}(\lambda + \log^2 n \log\log n)$, using $\mathcal{O}(\log n \log\log n)$ additional words of working space.*

To prove that Theorem 10 implies Theorem 11, we first show that we can always make $h = \mathcal{O}(\log n)$ without asymptotically increasing the size of the ISLP.

Lemma 12. *Given a d-ISLP G generating a string $T[1..n]$, it is possible to construct a d'-ISLP G' of size $\mathcal{O}(|G|)$ that generates T, for some $d' \leq d$, with height $h' = \mathcal{O}(\log n)$. The construction requires $\mathcal{O}((|G| + d)d\lceil d\log d/\log n\rceil)$ time and $\mathcal{O}(|G| + d\lceil d\log d/\log n\rceil)$ space.*

Proof. ISLPs are GSLPs: they allow rules of the form $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ of size $2+2t$, and a simple program of size $\mathcal{O}(t)$ writes the corresponding right-hand symbols (a sequence over $\{B_1, \ldots, B_t\}$) explicitly. Note that, if $k_1 \neq k_2$ for every special rule, then the corresponding GSLP is balanceable for sure, as no symbol in any output sequence can appear exactly once. If $k_1 = k_2$ for some special rule, instead, the output may have unique symbols $B_j^{i^0}$ or $B_j^{1^{c_j}}$. In this case we can split the rule at those symbols, in order to ensure that they do not appear in special rules, without altering the asymptotic size of the grammar. For example, $A \to B_1^{i^2} B_2^{i^0} B_3^{i^3} B_4^{i} B_5^{i^0}$ (i.e., $k_1 =$

$k_2 = i$ for some $i > 1$) can be converted into $A \to A_1 A_2$, $A_1 \to A_3 B_2$, $A_2 \to A_4 B_5$, $A_3 \to B_1^{i^2}$, $A_4 \to B_3^{i^3} B_4^i$. The case $k_1 = k_2 = 1$ corresponds to $A \to B_1 \cdots B_t$ and can be decomposed into normal binary rules within the same asymptotic size.

We can then apply Theorem 3. Note the exponents of the special rules $A \to x$ are retained in general, though some can disappear in the case $k_1 = k_2 = 1$. Thus, the parameter d' of the balanced ISLP satisfies $d' \leq d$.

The time to run the balancing algorithm is linear in $|G|$, except that we need to count, in the rules $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, how many occurrences of each nonterminal are produced. If we define

$$p_c(k) \;=\; \sum_{i=1}^{k} i^c, \qquad (1)$$

then B_j is produced $p_{c_j}(k_2) - p_{c_j}(k_1 - 1)$ times on the right-hand side of A.

Computing $p_c(k)$ straightforwardly takes time $\Omega(k)$, which may lead to a balancing time proportional to the length n of T. In order to obtain time proportional to the grammar size $|G|$, we need to process the rule for A in time proportional to its size, $\mathcal{O}(t)$. We show next how this can be done, by regarding $p_c(k)$ as a polynomial on k.

Proposition 13. *After a preprocessing time of $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, and within $\mathcal{O}(d \lceil d \log d / \log n \rceil)$ working space, we can compute any polynomial $p_c(k)$ in time $\mathcal{O}(d \lceil d \log d / \log n \rceil)$.*

Proof. An alternative formula [27][4] computes $p_c(k)$ using rational arithmetic (note $c \leq d$):

$$p_c(k) \;=\; k^c + \frac{1}{c+1} \cdot \sum_{j=0}^{c} \binom{c+1}{j} b_j \cdot k^{c+1-j}. \qquad (2)$$

The formula requires $\mathcal{O}(c) \subseteq \mathcal{O}(d)$ arithmetic operations once the numbers b_j are computed. Those b_j are the Bernoulli (rational) numbers. All the Bernoulli numbers from b_0 to b_d can be computed in $\mathcal{O}(d^2)$ arithmetic operations using the recurrence

$$\sum_{j=0}^{d} \binom{d+1}{j} b_j \;=\; 0,$$

from $b_0 = 1$. The numerators and denominators of the rationals b_j fit in $\mathcal{O}(j \log j) = \mathcal{O}(d \log d)$ bits,[5] so they can be operated in time $\mathcal{O}(\lceil d \log d / \log n \rceil)$ in a RAM machine with word size $\Theta(\log n)$. The total construction time of the Bernoulli numbers is then $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, and they can be maintained in $\mathcal{O}(d \lceil d \log d / \log n \rceil)$ space. ◻

Therefore, once we build the Bernoulli rationals b_j in advance, in time $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, the processing time for a rule of size $\mathcal{O}(t)$ is $\mathcal{O}(t\, d \lceil d \log d / \log n \rceil)$, which adds up to $\mathcal{O}(|G|\, d \lceil d \log d / \log n \rceil)$ for all the grammar rules. Storing the precomputed values b_j during construction requires $d \lceil d \log d / \log n \rceil$ extra space. ◻

[4] See also Wolfram Mathworld's `https://mathworld.wolfram.com/BernoulliNumber.html`, Eqs. (34) and (47).
[5] See `https://www.bernoulli.org`, sections "Structure of the denominator", "Structure of the nominator", and "Asymptotic formulas".

We now prove that we can always make $d = \mathcal{O}(\log n)$ without changing the size of the ISLP. From now on in the paper, we will disregard for simplicity the case $k_1 > k_2$ in the rules $A \rightarrow \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, as their treatment is analogous to that of the case $k_1 \leq k_2$.

Lemma 14. *If a d-ISLP G generates $T[1\,..\,n]$, then there is also a d'-ISLP G' of the same size that generates T, for some $d' \leq \log_2 n$.*

Proof. For any rule $A = \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, any $i \in [k_1\,..\,k_2]$, and any c_j, it holds that $n \geq |\mathtt{exp}(A)| \geq i^{c_j}$, and therefore $c_j \leq \log_i n$, which is bounded by $\log_2 n$ for $i \geq 2$. Therefore, if $k_2 \geq 2$, all the values c_j can be bounded by some $d' \leq \log_2 n$. A rule with $k_1 = k_2 = 1$ is the same as $A \rightarrow B_1 \cdots B_t$, so all values c_j can be set to 0 without changing the size of the rule. ☐

An even more special case.

The case $d = \mathcal{O}(1)$ deserves to be stated explicitly because it yields near-optimal substring extraction time, and because it already breaks the space lower bound $\Omega(\delta)$. We then plug $d = \mathcal{O}(1)$ and (per Lemma 12) $h = \mathcal{O}(\log n)$ in Theorem 10 to obtain the following result.

Corollary 15. *Let $T[1\,..\,n]$ be generated by a d-ISLP G, with $d = \mathcal{O}(1)$. Then, we can build in $\mathcal{O}(|G|)$ time and space a data structure of size $\mathcal{O}(|G|)$ that extracts any substring of T of length λ in time $\mathcal{O}(\lambda + \log n)$.*

Note that the corollary achieves $\mathcal{O}(\log n)$ access time for a single symbol. Verbin and Yu [46] showed that any data structure using space s to represent $T[1\,..\,n]$ requires time $\Omega(\log^{1-\epsilon} n / \log s)$ time, for *any* $\epsilon > 0$. Since even SLPs can use space $s = \mathcal{O}(\log n)$ on some texts, they cannot always offer access time $\mathcal{O}(\log^{1-\epsilon} n)$ for any constant ϵ. This restriction applies to even smaller grammars like RLSLPs and d-ISLPs for any d.

5.2 Data Structures

We now start to prove Theorem 10. In this subsection we focus on defining proper data structures that let us efficiently compute the length of the expansion of any prefix of the right-hand side of every rule

$$A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}.$$

This will be used in Section 5.3 to provide direct access to the content of $\mathtt{exp}(A)$. While our problem is easily solved by storing the $(k_2 - k_1 + 1) \cdot t$ cumulative lengths, we cannot afford that space. The challenge is to support these queries within space $\mathcal{O}(t)$, that is, proportional to the size of the rule.

Our key idea is that, though t can be large, there are only $d + 1$ distinct values c_j. We will exploit this because, per Lemma 14, d can be made $\mathcal{O}(\log n)$.

5.2.1 Navigating within a block

We start focusing on a single "block" $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, for fixed i. Our goal here is to efficiently compute the length of the expansion of $B_1^{i^{c_1}} \cdots B_r^{i^{c_r}}$ (i.e., a prefix of the block). Formally, we will compute the function

$$f_r(i) \;=\; \sum_{j=1}^{r} |\mathtt{exp}(B_j)| \cdot i^{c_j},$$

for any $r \in [1\,..\,t]$. We now show how to do this in time $\mathcal{O}(d)$, using $\mathcal{O}(t)$ space for rule A. We use two structures:

- We precompute an array $S_A[1\,..\,t]$ storing cumulative length information, as follows

  $$S_A[r] = \sum_{1 \le j \le r, c_j = c_r} |\mathtt{exp}(B_j)|.$$

  That is, $S_A[r]$ adds up the lengths, up to B_r, of the expansions of (only) those symbols that must be multiplied by i^{c_r}.
- A second array, $C_A[1\,..\,t]$, stores the values $c_1, \ldots, c_t$. We preprocess C_A to solve predecessor queries of the form

  $$\mathtt{pred}(A, r, c) = \max\{j \le r,\ C_A[j] = c\},$$

  that is, the latest occurrence of c in C_A to the left of position r, for every $c = 0, \ldots, d$.

To compute $f_r(i)$, we first calculate the values $r_c = \mathtt{pred}(A, r, c)$ for all c. We then evaluate $f_r(i)$ in $\mathcal{O}(d)$ time by adding up $S_A[r_c] \cdot i^c$, because $S_A[r_c]$ adds up all those $|\mathtt{exp}(B_j)|$, for $j \le r$, that must be multiplied by i^c in $f_r(i)$.

Example 1. *The left part of Figure 2 shows the arrays S_A and C_A of an example rule. The first (B^i), third (D^i), and sixth (E^i) symbols are raised to the power $i = i^1$ (i.e., $c_1 = c_3 = c_6 = 1$). Thus, $S_A[1] = 2 = |\mathtt{exp}(B)|$, $S_A[3] = 6 = S_A[1] + |\mathtt{exp}(D)|$, and $S_A[6] = 13 = S_A[3] + |\mathtt{exp}(E)|$. To compute $f_8(i)$, we will sum the coefficients of i^0, i^1, i^2, and i^3. The term to be multiplied by $i^0 = 1$ is $S_A[5]$, because $5 = \mathtt{pred}(A, 8, 0)$ is the last position in $[1\,..\,8]$ of a symbol raised to power i^0. In $S_A[5] = 14$ we have the sum of all the lengths that must be multiplied by i^0. Similarly, the term to be multiplied by i^1 is $S_A[6]$, because $6 = \mathtt{pred}(A, 8, 1)$ is the last position in $[1\,..\,8]$ of a symbol raised to power $i^1 = i$ (it is E^i). In $S_A[6] = 13$ we have the sum of all the lengths that must be multiplied by i. In $S_A[\mathtt{pred}(A, 8, 2) = 7]$ we have the total length 5 of the rules that must be multiplied by i^2 (which includes C^{i^2} and B^{i^2}), and in $S_A[\mathtt{pred}(A, 8, 3) = 8]$ we have the total length 3 of the rules to be multiplied by i^3 (which is just C^{i^3}). We then compute $f_8(i) = 14 + 13 \cdot i + 5 \cdot i^2 + 3 \cdot i^3$.*

We now show how to build S_A and how to precompute C_A so that the $d+1$ queries $\mathtt{pred}(A, r, c)$ are computed in $\mathcal{O}(d)$ time.

	1	2	3	4	5	6	7	8	9
S_A	2	3	6	7	14	13	5	3	18
C_A	1	2	1	0	0	1	2	3	0

$f_{r=8}(i) = 3i^3 + 5i^2 + 13i + 14$

$f_{r=9}(i) = 3i^3 + 5i^2 + 13i + 18$

$f^+(k) = \frac{9}{12}k^4 + \frac{38}{12}k^3 + \frac{117}{12}k^2 + \frac{304}{12}k$

Fig. 2 Data structures built for the ISLP rule $A \to \prod_{i=1}^{5} B^i C^{i^2} D^i EEE^i B^{i^2} C^{i^3} D$, with $|\texttt{exp}(B)| = 2$, $|\texttt{exp}(C)| = 3$, $|\texttt{exp}(D)| = 4$, and $|\texttt{exp}(E)| = 7$. On the right we show some of the polynomials that are computed with these data structures.

Building S_A.

Array S_A is built in time $\mathcal{O}(t)$ once all the lengths $|\texttt{exp}(\cdot)|$ have been computed, by traversing the nonterminals $B_1, \ldots, B_t$ in the rule of A while maintaining in an array $L[B_j]$ the last position of each distinct nonterminal B_j seen so far in the rule. Formally, the invariant is that, once we arrive at B_r, it holds $L[B] = \max\{i < r,\ B_i = B\}$ for all symbols $B \in \{B_1, \ldots, B_{r-1}\}$ (and $L[B] = 0$ if B has not appeared before B_r). We initialize $S_A[0] \leftarrow 0$ and, at step r, we fill $S_A[r] \leftarrow S_A[L[B_r]] + |\texttt{exp}(B_r)|$ and then set $L[B_r] \leftarrow r$ to restore the invariant. Storing L requires $\mathcal{O}(|G|)$ space at construction time; we use lazy initialization to avoid $\mathcal{O}(|G|)$ initialization time.

Preprocessing C_A.

We preprocess C_A as follows: we cut C_A into chunks of length $d+1$, and for each chunk $C_A[(d+1)\cdot j+1\,..\,(d+1)\cdot(j+1)]$ we store precomputed values $\texttt{pred}(A, (d+1)\cdot j, c)$ for all $c \in \{0, \ldots, d\}$. That is, each chunk stores the predecessor of every c to its left in C_A. Those precomputed values require only $\mathcal{O}(t)$ space because there are $d+1$ of them per chunk. They can be computed in $\mathcal{O}(t)$ time, on a left-to-right traversal of C_A, by using an array $L'[0\,..\,d]$ analogous to L, which at each position records the last occurrence seen so far of each value $c \in \{0, \ldots, d\}$. The values $L'[0\,..\,d]$ after processing each position $(d+1)\cdot j$ are precisely the values $\texttt{pred}(A, (d+1)\cdot j, c)$ we store with the chunk j.

Once this precomputation is completed, we answer queries as follows. To compute the values $r_c = \texttt{pred}(A, r, c)$ for all c, we find the chunk $j = \lceil r/(d+1) \rceil - 1$ where r belongs, initialize every $r_c = \texttt{pred}(A, (d+1)\cdot j, c)$ for every c (which is stored with the chunk j), and then scan the chunk prefix $C_A[(d+1)\cdot j+1\,..\,r]$ left to right, correcting every $r_c \leftarrow k$ if $c = C_A[k]$, for $k = (d+1)\cdot j+1\,..\,r$.

Algorithm 2 summarizes the whole process to compute $f_r(i)$, and the next lemma summarizes our result.

Lemma 16. *After $\mathcal{O}(|G|)$ precomputation time using $\mathcal{O}(|G| + d)$ working space, we obtain data structures that use $\mathcal{O}(|G|)$ space and can compute any $f_r(i)$ in time $\mathcal{O}(d)$.*

Algorithm 2 Computing $f_r(i)$ for nonterminal A, in time $\mathcal{O}(d)$

Input: Values i and r, arrays S_A and C_A, and precomputed values $\texttt{pred}(A, (d+1)j, c)$ for every j and c.
Output: The value $f_r(i)$.

```
1:  j ← ⌈r/(d+1)⌉ − 1    // the chunk where r belongs
2:  for c ← 0,...,d do    // collect last occurrence of each c to the left of the chunk
3:      r_c ← pred(A, (d+1)j, c)    // this is precomputed
4:  for k ← (d+1)j+1,...,r do    // update last occurrences within the chunk
5:      c ← C_A[k]
6:      r_c ← k
7:  s ← 0    // knowing the last occurrences of each c up to r, compute f_r(i)
8:  p ← 1
9:  for c ← 0,...,d do
10:     s ← s + S_A[r_c] · p
11:     p ← p · i
12: return s
```

5.2.2 Navigating between blocks

We now complete the calculation of the expansion length of any prefix of the rule of A. The following function adds up the expansion lengths of several whole blocks.

$$f^+(k) \;=\; \sum_{i=k_1}^{k} f_t(i),$$

that is, $f^+(k)$ is the cumulative sum of the length of the whole expressions $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ until $i = k$. The problem is, again, that we cannot afford the space of simply storing the $|k_2 - k_1| + 1$ values $f^+(k)$. We will instead compute $f^+(k)$ by reusing the same data structures we already store for $f_r(i)$.

Just as in Algorithm 2, for each $c = 0, \ldots, d$, we compute $t_c = \texttt{pred}(A, t, c)$ and $s_c = S_A[t_c]$, which is the total expansion length of the symbols that must be multiplied by i^c in the whole rule. We then multiply s_c by the sum of the factors i^c from $i = k_1$ to $i = k$, $s_c \cdot \sum_{i=k_1}^{k} i^c = s_c \cdot (p_c(k) - p_c(k_1 - 1))$, where $p_c(k)$ is defined in Eq. (1). Finally, we compute

$$f^+(k) \;=\; \sum_{c=0}^{d} s_c \cdot (p_c(k) - p_c(k_1 - 1)).$$

Since $p_c(k)$ is a polynomial on k of maximum degree $c + 1$ (see Eq. (2)), $f^+(k)$ is a polynomial on k of maximum degree $d + 1$.

Example 2. *Consider the ISLP of Lemma 6, defined by the rules* $S \rightarrow \prod_{i=1}^{k_2} A^i B$, $A \rightarrow \mathtt{a}$, *and* $B \rightarrow \mathtt{b}$. *The polynomials associated with the representation of the rule* S *are* $i^{c_1} = i$ *and* $i^{c_2} = 1$. *Then, we construct the auxiliary polynomials* $f_1(i) =$

$|\mathtt{exp}(A)|i^{c_1} = i$ *and* $f_2(i) = |\mathtt{exp}(A)|i^{c_1} + |\mathtt{exp}(B)|i^{c_2} = i+1$. *Finally, we construct the polynomial* $f^+(k) = \sum_{i=1}^{k} f_2(i) = \sum_{i=1}^{k}(i+1) = \frac{1}{2}k^2 + \frac{3}{2}k$. *Indeed, our calculation yields* $t_0 = 2$ *and* $t_1 = 1$, $S_A[1] = S_A[2] = 1$, $s_0 = s_1 = 1$, $s_0(p_0(k) - p_0(0)) = k$ *and* $s_1(p_1(k) - p_0(k)) = \frac{k(k+1)}{2}$, *and* $f^+(k)$ *is then* $k + \frac{k(k+1)}{2}$. *Figure 2 shows a more complex example.*

As shown in Proposition 13, we can compute all the Bernoulli polynomials, and then the coefficients of all the polynomials $p_c(k)$ in time $\mathcal{O}(d^2\lceil d\log d/\log n\rceil)$. This yields the following result.

Lemma 17. *Once the structures of Lemma 16 are built, we can build in time* $\mathcal{O}(d^2\lceil d\log d/\log n\rceil)$ *additional data structures that use* $\mathcal{O}(d\lceil d\log d/\log n\rceil)$ *space, which can compute any* $f^+(k)$ *in time* $\mathcal{O}(d\lceil d\log d/\log n\rceil)$.

5.3 Direct Access

Now that we can efficiently compute the expansion lengths of rule prefixes, we answer our most basic and elementary query: given the data structures of size $\mathcal{O}(|G|)$ defined in the previous sections, return the symbol $T[l]$ given an index l. Instead of using extra space to store precomputed values, we start the query process by computing all the polynomials $p_c(k)$, which are the same for every rule, in time $\mathcal{O}(d^2\lceil d\log d/\log n\rceil)$. With those polynomial coefficients precomputed, we can compute any $f^+(k)$, as well as any $f_r(i)$, for any rule in time $\mathcal{O}(d\lceil d\log d/\log n\rceil)$, using Lemmas 16 and 17.

For SLPs with derivation tree of height h, the problem is easily solved in $\mathcal{O}(h)$ time by storing the expansion size of every nonterminal, and descending from the root to the corresponding leaf using $|\mathtt{exp}(B)|$ to determine whether to descend to the left or to the right of every rule $A \to BC$. The general idea for d-ISLPs is similar, but now determining which child to follow in repetition rules is more complex (recall Alg. 1).

To access the l-th character of the expansion of $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ we first find the value i such that $f^+(i-1) < l \leq f^+(i)$ by using binary search (we let $f^+(i-1) = 0$ when $i = k_1$). Then, we find the value r such that $f_{r-1}(i) < l - f^+(i-1) \leq f_r(i)$ by using binary search on the subindex of the functions (we let $f_{r-1}(i) = 0$ for any i when $r = 1$). We then know that the search follows by B_r, with offset $l - f^+(i-1) - f_{r-1}(i)$ inside $|\mathtt{exp}(B_r)|^{i^{c_r}}$. The offset within B_r is then easily computed with a modulus. Algorithm 3 gives the details, using $\mathtt{succ}$ to denote the binary search in an ordered set (i.e., $\mathtt{succ}([x_1 \mathinner{..} x_m], l) = j$ iff $x_{j-1} < l \leq x_j$).

We carry out the first binary search so that, for every i we try, if $f^+(i) < l$ we immediately answer $i+1$ if $l \leq f^+(i+1)$; instead, if $l \leq f^+(i)$, we immediately answer i if $f^+(i-1) < l$. As a result, the search area is initially of length $|\mathtt{exp}(A)|$ and, if the answer is i, the search has finished by the time the search area is of length $\leq f^+(i) - f^+(i-1) = f_t(i)$. Thus, there are $\mathcal{O}(1 + \log(|\mathtt{exp}(A)|/f_t(i)))$ binary search steps. The second binary search is modified analogously so that it carries out $\mathcal{O}(1+\log(f_t(i)/(i^{c_r}|\mathtt{exp}(B_r)|)))$ steps. Summing the costs of both binary searches, and because $i^{c_r} \geq 1$, we have at most $\mathcal{O}(1+\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B_r)|))$ steps. As the search continues by B_r, the sum of binary search steps telescopes to $\mathcal{O}(h+\log n)$ on an ISLP

Algorithm 3 Direct access on d-ISLPs of height h in $\mathcal{O}((h + \log n + d)d)$ operations

Input: A variable A of an ISLP, and a position $l \in [1, |\mathtt{exp}(A)|]$.
Output: The character $\mathtt{exp}(A)[l]$.

1: **function** ACCESS(A, l)
2:     **if** $A \to a$ **then**   // found the leaf in the parse tree for $T[l]$
3:         **return** a
4:     **else if** $A \to BC$ **then**   // go left or right as on classic SLPs
5:         **if** $l \leq |\mathtt{exp}(B)|$ **then**
6:             **return** ACCESS(B, l)
7:         **else** $(l > |\mathtt{exp}(B)|)$
8:             **return** ACCESS$(C, l - |\mathtt{exp}(B)|)$
9:     **else** $(A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}})$ // find the proper descendant node
10:         $i \leftarrow \mathtt{succ}([f^+(k_1) \,..\, f^+(k_2)], l)$
11:         $l \leftarrow l - f^+(i-1)$
12:         $r \leftarrow \mathtt{succ}([f_1(i) \,..\, f_t(i)], l)$
13:         $l \leftarrow l - f_{r-1}(i)$
14:         **return** ACCESS$(B_r, (l - 1 \bmod |\mathtt{exp}(B_r)|) + 1)$

of height h: assume we traverse the ISLP from the initial symbol A_1 to the symbol A_h. The sum of the binary search steps is of the order of

$$\begin{aligned} &(1 + \log(|\mathtt{exp}(A_1)|/|\mathtt{exp}(A_2)|)) + (1 + \log(|\mathtt{exp}(A_2)|/|\mathtt{exp}(A_3)|)) \\ &\quad + \cdots + (1 + \log(|\mathtt{exp}(A_{h-1})|/|\mathtt{exp}(A_h)|)) \\ &= h + \log(|\mathtt{exp}(A_1)|/|\mathtt{exp}(A_h)|) \quad \leq \quad h + \log n. \end{aligned}$$

This yields our result for accessing a single symbol.

Lemma 18. *After the construction-time precomputation of Lemma 16 and the query-time preprocessing $\mathcal{O}(d^2\lceil \log d/\log n\rceil)$ of Lemma 17, we can access any symbol $T[l]$ in time $\mathcal{O}((h + \log n)\, d\lceil d \log d/\log n\rceil)$.*

Example 3. *We show how to access the $\mathtt{b}$ at position 14 of the string $T = \prod_{i=1}^{5} \mathtt{a}^i\mathtt{b}$. Consider the ISLP G and its auxiliary polynomials computed in Example 1. We start by computing $f^+(2) = 5$. As $l > 5$, we go right in the binary search and compute $f^+(4) = 14$. As $l \leq 14$ we go left, compute $f^+(3) = 9$ and find that $i = 4$. Hence, $T[l]$ lies in the expansion of $A^iB = A^4B$ at position $l_1 = l - f^+(i-1) = 5$. Then, we compute $f_1(4) = 4$. As $l_1 > 4$, we turn right and compute $f_2(4) = 5$, finding that $r = 2$. Hence, $T[l]$ lies in the expansion of $B^{i^0} = B^1$ at position $l_2 = l_1 - f_{r-1}(i) = 1$.*

5.4 Extracting Substrings

The last piece for proving Theorem 10 is to show how to extract substrings from T. Once we have accessed $T[l]$, it is possible to output the substring $T[l \,..\, l + \lambda - 1]$ in

Algorithm 4 Length-λ substring access on ISLPs of height h in $\mathcal{O}(h+\lambda)$ extra time

Input: A variable A of an ISLP, a position $l \in [1, |\texttt{exp}(A)|]$ and a length $\lambda > 0$.
Output: Outputs $\texttt{exp}(A)[l\,..\,l+\lambda-1]$ and returns the number of symbols it could not extract (if $l+\lambda-1 > |\texttt{exp}(A)|$).

```
 1: function EXTRACT(A, l, λ)
 2:     if A → a then      // found the leaf in the parse tree for T[l], first output
 3:         output a
 4:         λ ← λ − 1
 5:     else if A → BC then      // go left and/or right as in classic SLPs
 6:         if l ≤ |exp(B)| then
 7:             λ ← EXTRACT((B, l, λ))
 8:             if λ > 0 then      // go also right if there are symbols yet to output
 9:                 λ ← EXTRACT(C, 1, λ))
10:         else (l > |exp(B)|)
11:             λ ← EXTRACT(C, l − |exp(B)|, λ)
12:     else (A → ∏_{i=k1}^{k2} B_1^{i^{c1}} ··· B_t^{i^{ct}})      // find the first proper descendant node
13:         i ← succ([f⁺(k1) .. f⁺(k2)], l)
14:         l ← l − f⁺(i − 1)
15:         r ← succ([f_1(i) .. f_t(i)], l)
16:         l ← l − f_{r−1}(i)
17:         λ ← EXTRACT(B_r, (l − 1 mod |exp(B_r)|) + 1, λ)
18:         k ← ⌈l/|exp(B_r)|⌉ + 1
19:         while i ≤ k2 ∧ λ > 0 do      // iterate on the subsequent blocks
20:             while r ≤ t∧λ > 0 do      // iterate on the subsequent block symbols B_r
21:                 while k ≤ i^{c_r} ∧ λ > 0 do      // iterate within the copies of B_r
22:                     λ ← EXTRACT(B_r, 1, λ)
23:                     k ← k + 1
24:                 k ← 1
25:                 r ← r + 1
26:             r ← 1
27:             i ← i + 1
28:     return λ
```

$\mathcal{O}(\lambda+h)$ additional time, as we return from the recursion in Algorithm 3. We carry the parameter λ of the number of symbols (yet) to output, which is first decremented when we finally find the first symbol, $T[l]$, which we now output immediately. From that point, as we return from the recursion, we output up to λ following symbols and return the number of remaining symbols yet to output, until $\lambda = 0$. See Algorithm 4.

To analyze this algorithm, we note that it visits λ consecutive leaves in the parse tree, plus their ancestors. This is because the algorithm does not visit any node that is not an ancestor of a leaf that must be output: it first traverses towards $T[l]$, and then enters into a node only if there are remaining descendant leaves to visit (i.e., $\lambda > 0$). The ancestors of those leaves are composed of (i) the leftmost and rightmost paths that lead to $T[l]$ and $T[l+\lambda-1]$, and (ii) a set of complete subtrees between those

paths. The former contain up to $2h$ nodes; the latter include up to λ leaves and thus up to λ internal nodes, as there are no nodes of degree 1 in the parse tree.

The analysis also shows up in Algorithm 4. We distinguish two types of recursive calls. Initially the substring to extract is within one of the children of the grammar tree node, and thus only one recursive call is made. Those are the cases of lines 7, 11, and 17. The number of those calls is limited by the the height h of the grammar. Once we reach a node where the substring to extract spreads across more than one child, the λ symbols to output are distributed across more than one recursive call, ending in line 3 when outputting individual symbols. Those recursive calls form a tree with no unary paths and λ leaves, thus they add up to $\mathcal{O}(\lambda)$.

A final detail is that, in line 21 of Algorithm 4, we need to compute i^{c_r}. This can be done with modular exponentiation in time $\mathcal{O}(\log c_r) \subseteq \mathcal{O}(\log d)$. If $\lambda \geq |\mathtt{exp}(B_r^{i^{c_r}})|$, then the time $\mathcal{O}(\log c_r)$ to compute i^{c_r} is absorbed by the time to traverse the subtree of $B_r^{i^{c_r}}$.[6] Otherwise, $B_r^{i^{c_r}}$ is the rightmost symbol of the parse tree that we will traverse; this can happen h times only. This issue then adds $\mathcal{O}(h \log d)$ to the total time, which is absorbed by the time to reach $T[l]$; recall Lemma 18.

The total space for the procedure is $\mathcal{O}(h)$ for the recursion stack (which is unnecessary when returning a single symbol, since recursion can be eliminated in that case), plus $\mathcal{O}(d \lceil d \log d / \log n \rceil)$ for the precomputed Bernoulli rationals.[7] This concludes the proof of Theorem 10.

5.5 Composable Functions on Substrings

Other than extracting a text substring, we aim at computing more general functions on arbitrary ranges $T[p\,..\,q]$, in time that is independent of the length $q-p+1$ of the range. We show how to compute some functions that have been studied in the literature, focusing on *composable* ones.

Definition 6. *A function f from strings is* composable *if there exists a function g such that, for every pair of strings X and Y, it holds $f(X \cdot Y) = g(f(X), f(Y))$.*

We focus for now on two popular composable functions, which find applications for example on grammar-compressed suffix trees [14, 15].

Definition 7. *A* range minimum query (RMQ) *on $T[p\,..\,q]$ returns the leftmost position where the minimum value occurs in $T[p\,..\,q]$. Formally,*

$$\textsc{rmq}(T, p, q) \;=\; \min\{k \in [p\,..\,q] \,|\, \forall k' \in [p\,..\,q], T[k] \leq T[k']\}.$$

Definition 8. *A* next/previous smaller value query (NSV/PSV) *on $T[p\,..\,n]$/$T[1\,..\,p]$ and with value v finds the smallest/largest position following/preceding p with value at most v. If there is no such a position, it returns $n+1$/ 0. Formally,*

$$\textsc{nsv}(T, p, v) \;=\; \min(\{q \,|\, q \geq p, T[q] < v\} \cup \{n+1\}),$$

[6] Except if $i = 1$, where the result is simply 1 for any c_r.

[7] As these do not depend on the query, they could be precomputed at indexing time and be made part of the index, at a very modest increase in space.

Algorithm 5 Range minimum queries on SLPs of height h in $\mathcal{O}(h)$ time

Input: A variable A of an SLP and positions $1 \le p \le q \le |\mathtt{exp}(A)|$.
Output: Returns $\mathtt{rmq}(\mathtt{exp}(A)[p\mathinner{..}q])$ and the corresponding minimum value.

1: **function** $\textsc{rmq}(A, p, q)$
2: **if** $(p, q) = (1, |\mathtt{exp}(A)|)$ **then return** $\mathtt{rmq}(A)$ (which is precomputed)
3: **else if** $A \to BC$ **then** // first see if we go only left or only right
4: **if** $q \le |\mathtt{exp}(B)|$ **then return** $\textsc{rmq}(B, p, q)$
5: **else if** $p > |\mathtt{exp}(B)|$ **then return** $\textsc{rmq}(C, p - |\mathtt{exp}(B)|, q - |\mathtt{exp}(B)|)$
6: **else** $(p \le |\mathtt{exp}(B)| < q)$ // else compose a left suffix and a right prefix call
7: $\langle m_l, v_l \rangle \leftarrow \textsc{rmq}(B, p, |\mathtt{exp}(B)|)$
8: $\langle m_r, v_r \rangle \leftarrow \textsc{rmq}(C, 1, q - |\mathtt{exp}(B)|)$
9: **if** $v_l \le v_r$ **then return** $\langle m_l, v_l \rangle$
10: **else return** $\langle |\mathtt{exp}(B)| + m_r, v_r \rangle$

$$\textsc{psv}(T, p, v) \;=\; \max(\{q \mid q \le p, T[q] < v\} \cup \{0\}).$$

We show next how to efficiently solve those queries on ISLPs. We do not know how to compute other more complex functions, like Karp-Rabin fingerprints [21], on ISLPs. This will be addressed in Section 6, on the simpler RLSLPs.

5.5.1 Range Minimum Queries

Solving RMQs on an SLP G is simple thanks to composability. More precisely, what is composable is an extended function $f(X) = \langle m, v, \ell \rangle$ where $m = \textsc{rmq}(X, 1, |X|)$, $v = X[m]$, and $\ell = |X|$. Then, given $f(X) = \langle m_x, v_x, \ell_x \rangle$ and $f(Y) = \langle m_y, v_y, \ell_y \rangle$, it holds $f(X \cdot Y) = \langle m_x, v_x, \ell_x + \ell_y \rangle$ if $v_x \le v_y$, and $\langle \ell_x + m_y, v_y, \ell_x + \ell_y \rangle$ otherwise, which is computable in time $\mathcal{O}(1)$. We also compute $f(a) = \langle 1, a, 1 \rangle$ in $\mathcal{O}(1)$ time.

To compute RMQs on an SLP G, we first preprocess the grammar to store $f(\mathtt{exp}(A)) = \langle m, v, \ell \rangle$ for each nonterminal A, in the form of the pair $\mathtt{rmq}(A) = \langle m, v \rangle$ and the length $\ell = |\mathtt{exp}(A)|$. Thanks to the composability of f, this is easily built in $\mathcal{O}(|G|)$ time in a bottom-up traversal of the grammar.

To solve $\textsc{rmq}(T[p\mathinner{..}q])$ on the SLP, we descend from the root towards $T[p\mathinner{..}q]$ (guided by the stored expansion lengths $|\mathtt{exp}(A)|$) until finding a leaf (if $p = q$), or more typically a rule $A \to BC$ such that $T[p\mathinner{..}q] = \mathtt{exp}(B)[p'\mathinner{..}|\mathtt{exp}(B)|] \cdot \mathtt{exp}(C)[1\mathinner{..}q']$. At this point we split into two recursive calls, one computing $\textsc{rmq}$ on a suffix of $\mathtt{exp}(B)$ (a *suffix call*) and another on a prefix of $\mathtt{exp}(C)$ (a *prefix call*). By making the recursive calls return $\mathtt{rmq}(B)$ in $\mathcal{O}(1)$ time when the range spans the whole string $\mathtt{exp}(B)$, we ensure that those prefix/suffix calls perform only one further (nontrivial) recursive call per level, and thus the query is solved in $\mathcal{O}(h)$ time, traversing at most two root-to-leaf paths in the parse tree. Algorithm 5 shows the details.

To solve RMQs on ISLPs, we observe that the expansion of $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ always contains the same symbols. Further, the RMQ of $\mathtt{exp}(A)$ occurs always in the first block, $i = k_1$, and it depends essentially on the sequence $B_1 \cdots B_t$. To handle these rules, we preprocess them as follows. Let $\mathtt{rmq}(B_j) = \langle m_j, v_j \rangle$. Then, we build the string $v_1 \cdots v_t$ and precompute an RMQ data structure on it that answers

queries $\mathtt{rmq}_A(p,q) = \textsc{rmq}(v_1 \cdots v_t, p, q)$. It is possible to build such a data structure in $\mathcal{O}(t)$ time and bits of space, such that it answers queries in $\mathcal{O}(1)$ time [13], so this adds just $\mathcal{O}(|G|)$ time and bits to the grammar preprocessing cost. With this structure, we can simulate the extension of our $\mathtt{rmq}(A)$ precomputed pairs to any subsequence $B_a^{i^{c_a}} \cdots B_b^{i^{c_b}}$ of $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$: $\mathtt{rmq}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, a, b) = \langle m, v \rangle$, where $\textsc{rmq}(v_1 \cdots v_t, a, b) = m'$, $\mathtt{rmq}(B_{m'}) = \langle m'', v \rangle$, and $m = f_{m'-1}(i) + m''$. The time to compute this is dominated by the $\mathcal{O}(d)$ cost to compute $f_{m'-1}(i)$.

At query time, when we arrive at such a node A with limits p and q, we proceed as in lines 10–13 of Algorithm 3 to find the values i_p and r_p, and i_q and r_q, corresponding to p and q, respectively (just as we find i and r for l in Algorithm 3). There are several possibilities:

1. If $i_p = i_q$ and $r_p = r_q$, then p and q fall inside $\mathtt{exp}(B_{r_p}^{i^{c_{r_p}}})$. They may be both inside a single copy of $\mathtt{exp}(B_{r_p})$, in which case we continue with a single recursive call. Or they may span a (possibly empty) suffix of $\mathtt{exp}(B_{r_p})$, zero or more copies of $\mathtt{exp}(B_{r_p})$, and a (possibly empty) prefix of $\mathtt{exp}(B_{r_p})$. The query is then solved with at most two recursive calls on B_{r_p} (which are prefix/suffix calls), and the information on $\mathtt{rmq}(B_{r_p})$. We compose as explained those (up to) three results, and add $f^+(i_p - 1) + f_{r_p-1}(i_p)$ to the resulting position so as to place it within $\mathtt{exp}(A)$.
2. If $i_p = i_q$ and $r_p < r_q$, then we must also consider the subsequence $B_{r_p+1}^{i_p^{c_{r_p+1}}} \cdots B_{r_q-1}^{i_p^{c_{r_q-1}}}$, in case $r_q - r_p > 1$. This additional candidate to the RMQ is found with $\mathtt{rmq}(B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}, r_p + 1, r_q - 1)$, in time $\mathcal{O}(d)$ as explained.
3. If $i_p < i_q$, we must also add a suffix of of $B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}$, the whole $B_1^{(i_p+1)^{c_1}} \cdots B_t^{(i_p+1)^{c_t}}$ (if $i_q - i_p > 1$), and a prefix of $B_1^{i_q^{c_1}} \cdots B_t^{i_q^{c_t}}$ (if $i_q - i_p = 1$). All those are included with our simulation of queries $\mathtt{rmq}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, a, b)$.

Overall, we perform either one recursive call (when p and q are inside the same B_{r_p}), or two prefix/suffix recursive calls (for a suffix of B_{r_p} and a prefix of B_{r_q}). The analysis is then the same as for the SLPs, yielding time $\mathcal{O}(hd)$. This is in addition to (and dominated by) the $\mathcal{O}((h + \log n)d\lceil d \log d / \log n \rceil)$ time, plus the preprocessing time of $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, due to the binary searches needed to find i_p, i_q, r_p, and r_q, as for direct access (recall Lemma 18).

Theorem 19. *Let $T[1..n]$ be generated by a d-ISLP G of height h. Then, we can build in time $\mathcal{O}((|G| + d)d\lceil d \log d / \log n \rceil)$ and space $\mathcal{O}(|G| + d\lceil d \log d / \log n \rceil)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{rmq}(T, p, q)$ in time $\mathcal{O}((h + \log n + d)d\lceil d \log d / \log n \rceil)$, using $\mathcal{O}(h + d\lceil d \log d / \log n \rceil)$ additional words of working space.*

Since we can make both h and d be $\mathcal{O}(\log n)$ per Lemmas 12 and 14, we have the following corollary.

Corollary 20. *Let $T[1..n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G| + \log n) \log n \log \log n)$ and space $\mathcal{O}(|G| + \log n \log \log n)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{rmq}(T, p, q)$ in time $\mathcal{O}(\log^2 n \log \log n)$, using $\mathcal{O}(\log n \log \log n)$ additional words of working space.*

Finally, the following specialization is relevant, as for example it encompasses 1-ISLPs (which may break δ) and RLSLPs, and matches the analogous result on SLPs.

Corollary 21. *Let $T[1\,..\,n]$ be generated by a d-ISLP G with $d = \mathcal{O}(1)$. Then, we can build in time and space $\mathcal{O}(|G|)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query* $\textsc{rmq}(T,p,q)$ *in time $\mathcal{O}(\log n)$.*

5.5.2 Next/Previous Smaller Value

Let us consider query NSV; query PSV is analogous. NSV is composable if we extend it to function $f(X,v) = \langle p, \ell\rangle$, where $p = \textsc{nsv}(X,1,v)$ and $\ell = |X|$. If $f(X,v) = \langle p_x, \ell_x\rangle$ and $f(Y,v) = \langle p_y, \ell_y\rangle$, then $f(X \cdot Y) = \langle p, \ell_x + \ell_y\rangle$, where $p = p_x$ if $p_x \leq \ell_x$, else $p = \ell_x + p_y$ if $p_y \leq \ell_y$, and $p = \ell_x + \ell_y + 1$ otherwise. The composition takes $\mathcal{O}(1)$ time.

The procedure to compute $\textsc{nsv}(T,p,v)$ on an SLP is depicted in Algorithm 6. We reuse the precomputed pairs $\texttt{rmq}(A) = \langle m, v\rangle$ of RMQs, using $\texttt{rmq}(A).v$ to refer to v. Importantly, the algorithm uses that field to notice in constant time that the answer is not within $\texttt{exp}(A)$ (lines 2–3). In this case we say that the call to A *fails* (to find the answer within $\texttt{exp}(A)$). As for RMQs, the algorithm may perform two calls on $A \to BC$, which only happens when the call on B fails, but then the call on B is a suffix call and the call on C is a prefix call. Note that, in this asymmetric query with no right limit, a prefix call on C is a call on the whole $\texttt{exp}(C)$; we call it a *whole-symbol call*. As explained, those calls take $\mathcal{O}(1)$ time when they fail. Therefore,

- The suffix call starting from B, which finally fails, cannot branch again into two recursive calls at a symbol $A' \to B'C'$, because once the call on B' fails, the call on C' is a whole-symbol call, and this fails in constant time.
- The prefix call starting from C cannot branch again into two recursive calls at a symbol $A' \to B'C'$, because this occurs only if the call on B' fails. Since this is a whole-symbol call on B', it fails in constant time.

Since at most two paths are followed from the first branching into two calls, the total time is $\mathcal{O}(h)$.

To extend the algorithm to ISLPs we must consider, as for the case of RMQs, the special rules. Just as in that case, the answer to a query $\textsc{nsv}(\texttt{exp}(A),p,v)$ with $A \to \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ depends essentially on the smallest values of the nonterminal expansions, $\texttt{exp}(B_j)$. Let again $\texttt{rmq}(B_j) = \langle m_j, v_j\rangle$. We preprocess the string $v_1 \cdots v_t$ to solve queries $\textsc{nsv}(v_1 \cdots v_t, p)$. This preprocessing takes $\mathcal{O}(t \log t)$ time and $\mathcal{O}(t)$ space, and answers NSV queries in time $\mathcal{O}(\log^\epsilon t)$ for any constant $\epsilon > 0$ [42] (those are modeled as orthogonal range successor queries on a grid).

We can then simulate precomputed values $\texttt{nsv}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, p, v) = q$, where p refers to $B_p^{i^{c_p}} \cdots B_t^{i^{c_t}}$, with the value $\textsc{nsv}(v_1 \cdots v_t, p, v) = q'$ precomputed as explained, $\textsc{nsv}(B_{q'}, 1, v) = q''$ obtained with a recursive call, and $q = f_{q'-1}(i) + q''$. Note that the recursive call is for a whole symbol, and we are sure to find the answer inside it: if $\textsc{nsv}(v_1 \cdots v_t, p, v) = t+1$, we return $f_t(i) + 1$ without making any recursive call. At query time, after finding i_p and r_p as for RMQs, we have the following cases:

1. We may have to recurse on a nonempty suffix of B_{r_p}, finishing if we find the answer inside it. If not, there may be more copies of B_{r_p} ahead of position p, in which

Algorithm 6 Next smaller values on SLPs of height h in $\mathcal{O}(h)$ time

Input: A variable A of an SLP, position $1 \le p \le |\texttt{exp}(A)|$, and threshold v.
Output: The position $\text{NSV}(\texttt{exp}(A), p, v)$.

1: **function** $\text{NSV}(A, p, v)$
2: **if** $\texttt{rmq}(A).v \ge v$ **then** // whole symbols detect failure immediately
3: **return** $|\texttt{exp}(A)| + 1$
4: **else if** $A \to a$ **then**
5: **return** 1
6: **else if** $A \to BC$ **then**
7: **if** $p \le |\texttt{exp}(B)|$ **then** // first try to find the answer inside $\texttt{exp}(B)$
8: $p \leftarrow \text{NSV}(B, p, v)$
9: **if** $p \le |\texttt{exp}(B)|$ **then** // return the answer if found
10: **return** p
11: **return** $|\texttt{exp}(B)| + \text{NSV}(C, p - |\texttt{exp}(B)|, v)$ // else try on the whole C

case we either determine in constant time that there is no answer inside B_{r_p}, or we recurse with a whole-symbol call on B_{r_p} and find the answer inside it, thereby finishing.

2. If not finished, we may have to consider a block suffix $B_{r_p+1}^{i_p^{c_{r_p+1}}} \cdots B_t^{i_p^{c_t}}$. This is handled by computing $\texttt{nsv}(B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}, r_p + 1, v)$ as explained, possibly making a whole-symbol recursive call, only when we are sure to find the answer inside it.
3. If not, we may find the answer in the next block, $B_1^{(i_p+1)^{c_1}} \cdots B_t^{(i_p+1)^{c_t}}$, in the same way as in point 2. If we find no answer here, then there is no answer to NSV and we return $|\texttt{exp}(A)| + 1$.

Just as for the case of SLPs, we traverse only two paths along the process: even if now we have a sequence of more than two symbols (not just $A \to BC$), we are able to determine with a constant amount of `nsv` queries whether there is an answer to the right of the failing recursive call, and in which symbol we must recurse to find it. The main difference with the cost of RMQs is the $\mathcal{O}(\log^\epsilon t) \subseteq \mathcal{O}(\log^\epsilon |G|)$ time incurred to compute `nsv` queries, and the corresponding $\mathcal{O}(t \log t)$ construction time, which adds up to $\mathcal{O}(|G| \log |G|)$.

Theorem 22. *Let $T[1\,..\,n]$ be generated by a d-ISLP G of height h. Then, for any constant $\epsilon > 0$, we can build in time $\mathcal{O}(|G|(\log |G| + d\lceil d \log d / \log n \rceil) + d^2 \lceil d \log d / \log n \rceil)$ and space $\mathcal{O}(|G| + d\lceil d \log d / \log n \rceil)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\text{PSV}/\text{NSV}(T, p, v)$ in time $\mathcal{O}(h \log^\epsilon |G| + (h + \log n + d) d \lceil d \log d / \log n \rceil)$, using $\mathcal{O}(h + d\lceil d \log d / \log n \rceil)$ additional words of working space.*

Corollary 23. *Let $T[1\,..\,n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G| + \log n) \log n \log \log n)$ and space $\mathcal{O}(|G| + \log n \log \log n)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\text{PSV}/\text{NSV}(T, p, v)$ in time $\mathcal{O}(\log^2 n \log \log n)$, using $\mathcal{O}(\log n \log \log n)$ additional words of working space.*

Corollary 24. *Let $T[1\,..\,n]$ be generated by a d-ISLP G with $d = \mathcal{O}(1)$. Then, for any constant $\epsilon > 0$, we can build in time $\mathcal{O}(|G|\log|G|)$ and space $\mathcal{O}(|G|)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query* PSV/NSV(T,p,v) *in time $\mathcal{O}(\log n \log^{\epsilon}|G|)$.*

6 Revisiting RLSLPs

As pointed out in Proposition 4, RSLPs are equivalent to 0-ISLPs, because an ISLP rule $A \to \prod_{i=k_1}^{k_2} B^{i^0}$ corresponds exactly to the RLSLP rule $A \to B^{|k_2-k_1|+1}$. We can then apply Lemma 12 over any RLSLP to obtain an equivalent RLSLP of the same asymptotic size and height $\mathcal{O}(\log n)$. Once we count with a balanced version of any RLSLP, we can reuse Corollaries 15, 21, and 24, to obtain a similar result for RLSLPs. Note that we can improve those results because we do not need to preprocess the grammar to simulate the `rmq` and `nsv` queries on blocks, because in an RLSLP all the cases of run-length rules $A \to B^t$ fall inside the subcase 1 of RMQs and NSVs.

Corollary 25. *Let $T[1\,..\,n]$ be generated by a RLSLP G. Then, we can build in time and space $\mathcal{O}(|G|)$ data structures of size $\mathcal{O}(|G|)$ that (i) extract any substring $T[l\,..\,l+\lambda-1]$ in time $\mathcal{O}(\lambda+\log n)$, (ii) compute any query* RMQ(T,p,q) *in time $\mathcal{O}(\log n)$, and (iii) compute any query* PSV/NSV(T,p,v) *in time $\mathcal{O}(\log n)$.*

Those results on RLSLPs have already been obtained before [10, 15], but our solutions exploiting balancedness are much simpler once projected into the run-length rules. We now exploit the simplicity of RLSLPs to answer a wider range of queries on substrings, and show as a particular case how to compute Karp-Rabin fingerprints in logarithmic time; we do not know how to do that on general ISLPs.

6.1 More General Functions

We now expand our results to a wide family of composable functions that can be computed in $\mathcal{O}(\log n)$ time on top of balanced RLSLPs. We prove the following result.

Theorem 26. *Let f be a composable function from strings to a set of size $n^{\mathcal{O}(1)}$, computable in time t_f for strings of length 1, with its composing function g being computable in time t_g. Then, given an RLSLP G representing $T[1\,..\,n]$, there is a data structure of size $\mathcal{O}(|G|)$ that can be built in time $\mathcal{O}(|G|(t_f + t_g \log n))$ and that computes any $f(T[i\,..\,j])$ in time $\mathcal{O}(t_g \log n)$.*

Proof. By Theorem 3, we can assume G is balanced. We store the values $L[A] = |\texttt{exp}(A)|$ and $F[A] = f(\texttt{exp}(A))$ for every variable A, as arrays. These arrays add only $\mathcal{O}(|G|)$ extra space because the values in F fit in $\mathcal{O}(\log n)$-bit words. Let us overload the notation and use $f(A,i,j) = f(\texttt{exp}(A)[i\,..\,j])$. Algorithm 7 shows how to compute any $f(A,i,j)$; by calling it on the start symbol S of G we compute $f(T[i\,..\,j]) = f(S,i,j)$.

Just as for ISLPs, in the beginning we follow a single path along the derivation tree, with only one recursive call per argument A (lines 6, 8, and 17). The cost of those calls adds up to the height of the grammar, $\mathcal{O}(\log n)$. This path finishes at a leaf or at an internal node A where $\texttt{exp}(A)[i\,..\,j]$ spans more than one child of A in the derivation tree, in which case we may perform two recursive calls. Note that in the

Algorithm 7 Computation of general string functions in RLSLPs in $\mathcal{O}(\log n)$ steps

Input: A variable A of an RLSLP (with its arrays L and F as global variables), and two positions $1 \le i \le j \le |\texttt{exp}(A)|$.

Output: $f(\texttt{exp}(A)[i \mathinner{..} j])$.

```
1:  function F(A, i, j)
2:      if (i, j) = (1, |exp(A)|) then    // whole symbols solved in constant time
3:          return F[A]
4:      else if A → BC then     // try to recurse only left or right
5:          if j ≤ |exp(B)| then
6:              return F(B, i, j)
7:          else if |exp(B)| < i then
8:              return F(C, i − |exp(B)|, j − |exp(B)|)
9:          else (i ≤ |exp(B)| < j)     // compose left suffix and right prefix calls
10:             f_l ← F(B, i, |exp(B)|)
11:             f_r ← F(C, 1, j − |exp(B)|)
12:             return g(f_l, f_r)
13:     else (A → B^t)     // run-length rule spanning from t' to t''
14:         t' ← ⌈i/|exp(B)|⌉
15:         t'' ← ⌈j/|exp(B)|⌉
16:         if t' = t'' then     // still recurse on only one symbol
17:             return F(B, i − (t' − 1) · |exp(B)|, j − (t' − 1) · |exp(B)|)
18:         f_l ← F(B, i − (t' − 1) · |exp(B)|, |exp(B)|)     // left suffix call
19:         f_r ← F(B, 1, j − (t'' − 1) · |exp(B)|)     // right prefix call
20:         Compute f_c(t'' − t' − 1) using the recurrence     // many whole symbols
```

$$f_c(k) \leftarrow \begin{cases} f(\varepsilon) & \text{if } k = 0; \\ F[B] & \text{if } k = 1; \\ g(f_c(k/2), f_c(k/2)) & \text{if } k \text{ is even}; \\ g(F[B], f_c(k-1)) & \text{if } k \text{ is odd}. \end{cases}$$

```
21:         return g(g(f_l, f_c(t'' − t − 1)), f_r)     // compose left, middle, right
```

only places where this may occur (lines 10–11 and 18–19) those recursive calls will be prefix/suffix calls (i.e., either $i = 1$ or $j = L[A]$ when we call $\textsc{f}(A, i, j)$). We now focus on bounding the cost of prefix/suffix calls.

We define $c(A)$ as the highest cost to compute $f(A, i, L[A])$ or $f(A, 1, j)$ over any i and j (i.e., the cost of prefix/suffix calls), charging 1 to the number of calls to function $\textsc{f}$ and t_g to each invocation to function g. We assume for simplicity that $t_g \ge 1$ and prove by induction that $c(A) \le (1 + 2t_g)d(A) + 2t_g \log|\texttt{exp}(A)|$, where $d(A)$ is the distance from A to its deepest descendant leaf in the derivation tree. This certainly holds in the base case of leaves, where $d(A) = 1$; it is included in lines 2–3.

In the inductive case of rules $A \to BC$ (lines 4–12), we note that there can be two calls to $\textsc{f}$, but in prefix/suffix calls one of those calls spans the whole symbol—line 10

in a prefix call or line 11 in a suffix call. Calls that span the whole symbol finish in line 3 and therefore cost just 1. Therefore, we have $c(A) \leq 1 + \max(c(B), c(C)) + t_g$, which by induction is

$$\begin{aligned}
c(A) &\leq 1 + \max(c(B), c(C)) + t_g \\
&\leq 1 + t_g + \max((1{+}2t_g)d(B) + 2t_g \log|\mathtt{exp}(B)|, (1{+}2t_g)d(C) + 2t_g \log|\mathtt{exp}(C)|) \\
&\leq (1 + 2t_g)(1 + \max(d(B), d(C))) + 2t_g \log\max(|\mathtt{exp}(B)|, |\mathtt{exp}(C)|) \\
&\leq (1 + 2t_g)d(A) + 2t_g \log|\mathtt{exp}(A)|).
\end{aligned}$$

In the inductive case of rules $A \rightarrow B^t$ (lines 13–21), a similar situation occurs in lines 18–19: only one of the two recursive calls is nontrivial. Therefore, it holds $c(A) \leq 1 + c(B) + 2t_g \log t + 2t_g$, where the term $2t_g \log t$ comes from the recursive procedure to compute $f_c(t'' - t' - 1)$ in line 20; the logarithm is in base 2. Because $t = |\mathtt{exp}(A)|/|\mathtt{exp}(B)|$, by induction we have

$$\begin{aligned}
c(A) &\leq 1 + c(B) + t_g(2 + 2\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|)) \\
&\leq 1 + (1 + 2t_g)d(B) + 2t_g \log|\mathtt{exp}(B)| + 2t_g(1 + \log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|)) \\
&= (1 + 2t_g)(1 + d(B)) + 2t_g \log|\mathtt{exp}(A)| \;=\; (1 + 2t_g)d(A) + 2t_g \log|\mathtt{exp}(A)|.
\end{aligned}$$

Therefore, the procedure costs $c(A) = (1{+}2t_g)d(A){+}2t_g \log|\mathtt{exp}(A)| = \mathcal{O}(t_g{\cdot}\log n)$ from the nonterminal A in which the single path splits into two.

Arrays L and F can be precomputed in time $\mathcal{O}(|G|(t_f + t_g \log n))$ via a postorder traversal of the grammar tree. We compute f for every distinct individual symbol and g for each distinct nonterminal A, whose children have by then their L and F entries already computed. In the case of rules $A \rightarrow B^t$, the entry $F[A]$ can be computed in time $\mathcal{O}(t_g \log t)$ with the same mechanism used in line 20 of Algorithm 7. ☐

We show in the next section how to use this result to compute a more complicated function, which in particular we do not know how to compute efficiently on ISLPs.

6.2 Application: Karp-Rabin Fingerprints

Given a string $T[1\,..\,n]$, a suitable integer c, and a prime number $\mu \in \mathcal{O}(n)$, the Karp-Rabin fingerprint [21] of $T[i\,..\,j]$, for $1 \leq i \leq j \leq n$, is defined as

$$\kappa(T[i\,..\,j]) \;=\; \left(\sum_{k=i}^{j} T[k] \cdot c^{k-i}\right) \bmod \mu.$$

Computation of fingerprints of text substrings from their grammar representation is a key component of various compressed text indexing schemes [10]. While it is known how to compute it in $\mathcal{O}(\log n)$ time using $\mathcal{O}(|G|)$ space on an RLSLP G [10, App. A], we show now a much simpler procedure that is an application of Theorem 26.

Note that, for any split position $p \in [i\,..\,j-1]$, it holds

$$\kappa(T[i\,..\,j]) \;=\; \left(\kappa(T[i\,..\,p]) + \kappa(T[p+1\,..\,j]) \cdot c^{p-i+1}\right) \bmod \mu. \tag{3}$$

We use this property as a basis for the efficient computation of fingerprints on RLSLPs.

Theorem 27 (cf. [6, 10])**.** *Given an RLSLP G representing $T[1..n]$ and a Karp-Rabin fingerprint function κ, there is a data structure of size $\mathcal{O}(|G|)$ that can be built in time $\mathcal{O}(|G|\log n)$ and computes fingerprints of arbitrary substrings of T in $\mathcal{O}(\log n)$ time.*

Proof. Let $f(X) = \langle \kappa(X), c^{|X|}\rangle$ be the function f to apply Theorem 26. We then define

$$g(\langle \kappa_x, c_x\rangle, \langle \kappa_y, c_y\rangle) = \langle (\kappa_x + \kappa_y \cdot c_x) \bmod \mu, (c_x \cdot c_y) \bmod \mu\rangle,$$

which can be computed in time $t_g = \mathcal{O}(1)$.

It is easy to see that, by Eq. (3), $f(XY) = \langle \kappa(XY), c^{|XY|}\rangle = \langle (\kappa(X) + \kappa(Y) \cdot c^{|X|}) \bmod \mu, (c^{|X|} \cdot c^{|Y|}) \bmod \mu\rangle = g(\langle \kappa(X), c^{|X|}\rangle, \langle \kappa(Y), c^{|Y|}\rangle) = g(f(X), f(Y))$. Therefore, application of Theorem 26 leads to a procedure that computes $f(T[i..j]) = \langle \kappa(T[i..j]), c^{j-i+1} \bmod \mu\rangle$ in time $\mathcal{O}(\log n)$ and using $\mathcal{O}(|G|)$ extra space. □

7 Conclusions

We have generalized a recent result by Ganardi et al. [17], which shows how to balance any SLP while maintaining its asymptotic size. Our generalization, called GSLP, allows rules of the form $A \to x$, where x is a program that generates the actual (possibly much longer) right-hand side. While we believe that this general result can be of wide interest to balance many kinds of generalizations of SLPs, we demonstrate its usefulness on a particular generalization of SLPs we call Iterated SLPs (ISLPs). ISLPs are the first representation offering polylogarithmic-time access to the string symbols (thanks to balancing), while at the same time outperforming the well-known space measure δ [10, 43], based on substring complexity, by a large $\mathcal{O}(\sqrt{n})$ factor. A certain subclass of ISLPs achieves $\mathcal{O}(\log n)$ access time, which is nearly optimal [46]. This subclass includes Run-Length SLPs (RLSLPs), which extend SLPs just with the rule $A \to B^t$. Balancing allows us computing a wide class of substring queries on RLSLPs.

Bannai et al. [4] made another interesting use of our result on balancing RLSLPs (in its conference version [41]), to show that a certain restriction on Lempel-Ziv parsing that enables access to an arbitrary position in $\mathcal{O}(\log n)$ predecessor queries, always reaches size $\mathcal{O}(g_{rl})$. The size $z^* \geq z$ of such parsing is then a new accessible repetitiveness measure that outperforms RLSLPs. The same string family that was used to show that z can be $o(g_{rl})$ [7] serves to show that z^* can be $o(g_{rl})$; therefore z^* is a new accessible measure strictly better than g_{rl}.

The variables produced by the programs x in our GSLP rules must appear twice for our balancing to work. It is possible to lift this restriction by cutting the rules $A \to x$ so as to isolate, if needed, variables that appear once in the output of x. For this to work, the computation model must allow splitting the program x as well, so that the combined size of the new programs does not double along the process. While this seems not possible in Turing-complete models, there could be interesting generalizations of ISLPs that can be balanceable in this way. Actually, we note that the balancing procedure only needs the special variables to be endpoints of the SC-paths.

This is ensured if variables appear twice, but there could be more relaxed conditions to ensure balancing that might be worth exploring.

Several questions remain open on ISLPs. One is about the cost to find the smallest ISLP that generates a given text T; we conjecture the problem is NP-hard as it is for plain SLPs [9] and RLSLPs [22]. Indeed, since RLSLPs correspond exactly to what we called 0-ISLPs, finding the smallest 0-ISLP is NP-hard. It is open to extend this result to d-ISLPs for other values of d (this parameter limits the complexity of the ISLP).

A second question is whether we can build an index based on ISLPs that offers efficient pattern matching. While ISLPs support random access to the text, the typical path followed for SLPs [11] and for RLSLPs [10, App. A] cannot be directly applied for ISLPs, because iteration rules, which are of size $\mathcal{O}(t)$, would require indexing $\Theta(kt)$ positions. Computing Karp-Rabin fingerprints [21] on text substrings, which can be done in logarithmic time on SLPs and RLSLPs and enable substring equality and longest common prefix computations on T, is also challenging on general ISLPs.

Declarations

Funding

All authors were funded with Basal Funds FB0001 and AFB240001, ANID, Chile. F.O. was funded by scholarship ANID-Subdirección de Capital Humano/Doctorado Nacional/2021-21210579, ANID, Chile. C.U. was funded by scholarship ANID-Subdirección de Capital Humano/Doctorado Nacional/2021-21210580, ANID, Chile.

Competing interests

All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Ethics approval

Not applicable.

Consent to participate

All authors agreed to participate in this work.

Consent for publication

All authors agree for the publication of this work.

Availability of data and materials

Not applicable.

Code availability

Not applicable.

Authors contributions

The three authors, G.N., F.O., and C.U. participated equally in the conception, proofs, and writing of the paper.

References

[1] Akagi T, Funakoshi M, Inenaga S (2023) Sensitivity of string compressors and repetitiveness measures. Information and Computation 291:104999

[2] Allouche JP, Shallit J (1999) The ubiquitous prouhet-thue-morse sequence. In: Ding C, Helleseth T, Niederreiter H (eds) Sequences and their Applications. Springer London, pp 1–16

[3] Bannai H, Funakoshi M, I T, et al (2021) A separation of γ and b via Thue-Morse words. In: Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE), pp 167–178

[4] Bannai H, Funakoshi M, Hendrian D, et al (2024) Height-bounded lempel-ziv encodings. CoRR 2403.08209

[5] Bille P, Landau GM, Raman R, et al (2015) Random access to grammar-compressed strings and trees. SIAM Journal on Computing 44(3):513–539

[6] Bille P, Gørtz IL, Cording PH, et al (2017) Fingerprints in compressed strings. Journal of Computer and System Sciences 86:171–180

[7] Bille P, Gagie T, Gørtz IL, et al (2018) A separation between RLSLPs and LZ77. Journal of Discrete Algorithms 50:36–39

[8] Burrows M, Wheeler D (1994) A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation

[9] Charikar M, Lehman E, Liu D, et al (2005) The smallest grammar problem. IEEE Transactions on Information Theory 51(7):2554–2576

[10] Christiansen AR, Ettienne MB, Kociumaka T, et al (2020) Optimal-time dictionary-compressed indexes. ACM Transactions on Algorithms 17(1):article 8

[11] Claude F, Navarro G, Pacheco A (2021) Grammar-compressed indexes with logarithmic search time. Journal of Computer and System Sciences 118:53–74

[12] Fici G, Romana G, Sciortino M, et al (2023) On the impact of morphisms on BWT-runs. In: Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM), pp 10:1–10:18

[13] Fischer J, Heun V (2011) Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2):465–492

[14] Fischer J, Mäkinen V, Navarro G (2009) Faster entropy-bounded compressed suffix trees. Theoretical Computer Science 410(51):5354–5364

[15] Gagie T, Navarro G, Prezza N (2020) Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. Journal of the ACM 67(1):article 2

[16] Gallant JK (1982) String compression algorithms. PhD thesis, Princeton University

[17] Ganardi M, Jeż A, Lohrey M (2021) Balancing straight-line programs. J ACM 68(4):article 27

[18] Giuliani S, Inenaga S, Lipták Z, et al (2021) Novel results on the number of runs of the Burrows-Wheeler-transform. In: Proc. Theory and Practice of Computer Science (SOFSEM), pp 249–262

[19] Giuliani S, Inenaga S, Lipták Z, et al (2023) Bit catastrophes for the burrows-wheeler transform. In: Proc. Developments in Language Theory (DLT), pp 86–99

[20] Karhumäki J (1983) On cube-free ω-words generated by binary morphisms. Discrete Applied Mathematics 5(3):279–297

[21] Karp RM, Rabin MO (1987) Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2):249–260

[22] Kawamoto A, I T, Köppl D, et al (2024) On the hardness of smallest RLSLPs and collage systems. In: Proc. Data Compression Conference (DCC), pp 243–252

[23] Kempa D, Kociumaka T (2020) Resolution of the Burrows-Wheeler transform conjecture. In: Proc. 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS), pp 1002–1013

[24] Kempa D, Kociumaka T (2023) Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In: Proc. IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS), pp 1877–1886

[25] Kempa D, Prezza N (2018) At the roots of dictionary compression: String attractors. In: Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC), pp 827–840

[26] Kempa D, Saha B (2022) An upper bound and linear-space queries on the LZ-End parsing. In: Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), pp 2847–2866

[27] Knuth DE (2001) Johann Faulhaber and sums of powers. In: Selected Papers on Discrete Mathematics. Cambridge University Press

[28] Kociumaka T, Navarro G, Prezza N (2023) Towards a definitive compressibility measure for repetitive sequences. IEEE Transactions on Information Theory 69(4):2074–2092

[29] Kociumaka T, Navarro G, Olivares F (2024) Near-optimal search time in δ-optimal space, and vice versa. Algorithmica 86(4):1031–1056

[30] Kreft S, Navarro G (2013) On compressing and indexing repetitive sequences. Theoretical Computer Science 483:115–133

[31] Lempel A, Ziv J (1976) On the complexity of finite sequences. IEEE Transactions on Information Theory 22(1):75–81

[32] Lindenmayer A (1968) Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. Journal of Theoretical Biology 18(3):280–299

[33] Lindenmayer A (1968) Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. Journal of Theoretical Biology 18(3):300–315

[34] Mantaci S, Restivo A, Sciortino M (2003) Burrows–Wheeler transform and Sturmian words. Information Processing Letters 86(5):241–246

[35] Navarro G (2021) Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Computing Surveys 54(2):article 29

[36] Navarro G (2021) Indexing highly repetitive string collections, part II: Compressed indexes. ACM Computing Surveys 54(2):article 26

[37] Navarro G, Urbina C (2021) On stricter reachable repetitiveness measures. In: Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE), pp 193–206

[38] Navarro G, Urbina C (2023) L-systems for measuring repetitiveness. In: Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM), p article 14

[39] Navarro G, Urbina C (2024) Iterated straight-line programs. In: Proc. 16th Latin American Theoretical Informatics (LATIN), pp 66–80

[40] Navarro G, Ochoa C, Prezza N (2021) On the approximation ratio of ordered parsings. IEEE Transactions on Information Theory 67(2):1008–1026

[41] Navarro G, Olivares F, Urbina C (2022) Balancing run-length straight-line programs. In: Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE), pp 117–131

[42] Nekrich Y, Navarro G (2012) Sorted range reporting. In: Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT), LNCS 7357, pp 271–282

[43] Raskhodnikova S, Ron D, Rubinfeld R, et al (2013) Sublinear algorithms for approximating string compressibility. Algorithmica 65(3):685–709

[44] Sipser M (2012) Introduction to the Theory of Computation. Cengage Learning

[45] Storer JA, Szymanski TG (1982) Data compression via textual substitution. Journal of the ACM 29(4):928–951

[46] Verbin E, Yu W (2013) Data structure lower bounds on random access to grammar-compressed strings. In: Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM), pp 247–258