Navigating Forest Straight-Line Programs in constant time

Carl Philipp Reh[1] and Kurt Sieber[1]

Universität Siegen, Germany
{reh,sieber}@informatik.uni-siegen.de

Abstract. We present a data structure of linear size that allows to perform navigation steps and subtree equality checks in grammar-compressed forests in constant time. Navigation steps include going to the parent, to the left/right neighbor or to the first/last child.

Keywords: grammar-compressed forests · forest straight-line programs · algorithms for compressed forests.

1 Introduction

SLPs (straight-line programs) are context-free grammars that produce a single string. They can use nonterminals to identify repetition and therefore be much smaller than the string that they represent. For example, the SLP with $A_0 \to ab$ and $A_{i+1} \to A_i A_i$ for $1 \leq i < 10$ is a succinct representation of the string $(ab)^{1024}$. *Compression algorithms* try to find a succinct representation for a given input string. Some of them produce SLPs directly, while the output of others can be efficiently translated into SLPs [9]. Since SLPs can compress exponentially, i.e. an SLP of size n might produce a string of size $\Theta(2^n)$, answering a query about the represented string by uncompressing the SLP has exponential runtime. It is however often possible to implement algorithms that do better. A simple example is to print the character at a given position, which can be easily done in time $\mathcal{O}(h)$ where h is the height of the syntax tree of the SLP. Another example is navigating on the string of an SLP: We can start at the first position, go to the next position, go to the previous position or print the character at the current position. Each of these operations takes constant time if we allow linear time preprocessing. This was shown in [12], which extended a result from [8]. The authors use the Word RAM model, i.e. operations on integers whose number of bits is logarithmic in the length of the input (the SLP) require constant time, which is also the model we assume throughout the paper.

SLPs have been extended to TSLPs (tree straight-line programs) which allow to compress trees (see [11] for a survey). Instead of strings, a TSLP can store trees and *tree contexts*, i.e. trees with a single hole x. For example, a very tall tree $\underbrace{a(\cdots a(}_{2^n \text{ times}} b) \cdots)$ can be compressed by first applying the tree context $a(x)$ 2^n times to itself, which requires n productions, and then replacing x with b. Navigation has been extended to TSLPs in [12], where a navigation step can go to a specific child, to the parent node, or print the character at the current node. Each of these operations requires constant time after linear time preprocessing. Another operation has also been introduced in [12]: After navigating to two (different) nodes in the tree, we can ask if the subtrees at these nodes are equal. This problem appears in several contexts, see for example [5]. Adding the ability to do subtree equality checks however comes at a cost: First, the preprocessing time is now polynomial instead of linear, but the data structure that is precomputed still has linear size. Second, navigation steps now need to keep track of how deep into the tree they went, which requires integers whose number of bits is logarithmic in the tree size. We still refer to this as being constant time, which was also done in [12] and [2]. Grammar-compressed multigraphs are another data structure for which constant time traversal was implemented [13].

A shortcoming of TSLPs is that they can only compress vertically but not horizontally. This is a limiting factor because a lot of tree-like documents (i.e. Wikipedia articles) tend to be very wide but not very tall. A workaround is to transform a tree into its *fcns* (first-child next-sibling) encoding. This is basically a head-tail representation of a tree, e.g. a tree $a(bcd)$ is represented as $a(b(\perp, c(\perp, d(\perp, \perp))), \perp)$, where $\perp$ means that there is no first child or next sibling. This way, TSLP compression can work better because horizontal repetition is turned into vertical repetition. However, it can be tricky to design algorithms that try to answer queries *on the original tree* using only the TSLP for the tree's fcns encoding. For example, it is mentioned in [12] that using the fcns we can support in constant time the navigation operations of going to the first child or to the next sibling. But how we can go to the parent or to the last child (if it is at all possible) in constant time is unclear. A better way to deal with arbitrarily wide trees is to compress them more directly. An FSLP (forest straight-line program), introduced in [7], is basically an extension of a TSLP: It can compress vertically using tree contexts like a TSLP can, but it can also compress horizontally. For example, the tree $a(c^{1024} \cdots a(c^{1024}(b)) \cdots)$, which is a slight variation of the previous example, has a very direct succinct representation: We can compress c^{1024} and thus also the contexts $a(c^{1024}x)$.

In this work, we first introduce FSLP navigation without subtree equality checks, which only requires linear preprocessing time. The allowed operations are: go to the first/last child, the next/previous sibling or to the parent node, or print the current character. Then, we add an extension that allows subtree equality checks but requires polynomial preprocessing time, while still producing a structure of linear size. In both cases, we achieve the same results as the ones previously shown for TSLPs in [12]. While implementing our data structures and algorithms, we use SLP navigation as a black box instead of extending its structure like it was done in [12].

Another formalism for forest compression that is very similar to FSLPs are Top Dags [1]. However, the most basic trees that can be represented with Top Dags are of the form $a(b)$ and the most basic contexts are $a(b(x))$. This leads to cases where the smallest Top Dag is by a factor of the alphabet size larger than the smallest FSLP [7]. Since it is also possible to transform Top Dags into equivalent FSLPs in linear time [7], and we allow linear time preprocessing, all the results of this paper basically extend to Top Dags.

2 Preliminaries

For a string $w = a_1 \dots a_n \in \Theta^n$ with $n \geq 0$ over some alphabet Θ we write $w[i] = a_i$ for $1 \leq i \leq n$, $w[i : j] = a_i \dots a_j$ for $1 \leq i \leq j \leq n$, $w[: i] = w[1 : i]$ and $w[i :] = w[i : n]$. In [7] the authors unified the view of several SLP-like structures, which we are also going to use in this work. What was formerly called SLP is now called SSLP (string straight-line program) which frees up the name SLP to be used for something else: An SLP can compress *any* expression by giving names to common subexpressions. This unifies the view of SSLPs, TSLPs, FSLPs and Top Dags.

2.1 Algebras and straight-line programs

An *algebra* $\mathcal{A} = (\mathcal{U}, \mathcal{I})$ consists of a universe (the carrier set) $\mathcal{U}$ and a set of operations $\mathcal{I}$, where each $f \in \mathcal{I}$ is a partial function $f\colon \mathcal{U}^k \to \mathcal{U}$ for some $k \in \mathbb{N}$. Instead of allowing partial functions we could also have defined multi-sorted algebras (see for example [4]), but we feel that at least in this paper this would have unnecessarily complicated the notation. The *expressions* $\mathcal{E}$ over $\mathcal{A}$ consist of terms over the elements from $\mathcal{I}$, i.e. if $f \in \mathcal{I}$ and f is k-ary then $f(e_1, \dots, e_k) \in \mathcal{E}$ for all $e_1, \dots, e_k \in \mathcal{E}$. Evaluating an expression $e \in \mathcal{E}$ is written as $[\![e]\!]_{\mathcal{A}}$ and is defined in the usual way. For example, consider $\mathcal{U} = \mathbb{N}$ and $\mathcal{I} = \{+, 1\}$, where $+\colon \mathbb{N}^2 \to \mathbb{N}$ and $1 \in \mathbb{N}$, with the usual meaning. The expressions are $\mathcal{E} = \{1, +(1,1), +(+(1,1),1), +(1,+(1,1)), \dots\}$, and, for example, $[\![+(+(1,1),1)]\!]_{\mathcal{A}} = (1+1)+1 = 3$.

For a set of variables V we write $\mathcal{E}_V$ for the set of *expressions with variables* over $\mathcal{A}$. A variable can occur anywhere where a nullary function symbol can occur. For example, if $X, Y \in V$ then $+(+(1,X),Y) \in \mathcal{E}_V$.

A *straight-line program* (SLP for short) over $\mathcal{A}$ is a tuple $G = (V, \mathrm{rhs})$, where V is a finite set of variables and $\mathrm{rhs}\colon V \to \mathcal{E}_V$ is the right-hand side mapping. The relation $\{(A,B) \in V^2 \mid B \text{ appears in } \mathrm{rhs}(A)\}$ must be acyclic. This way, we can transform every expression from $e \in \mathcal{E}_V$ into one from $\mathcal{E}$ by recursively applying rhs. The result can then be evaluated in the algebra $\mathcal{A}$, which we denote with $[\![e]\!]_G \in \mathcal{U}$. For example, if $V = \{X, Y\}$ with $\mathrm{rhs}(X) = +(1,Y)$ and $\mathrm{rhs}(Y) = +(1,1)$, then $[\![+(X,Y)]\!]_G = [\![+(+(1,+(1,1)),+(1,1))]\!]_{\mathcal{A}} = 5$. The *size of an expression* is $|f(e_1,\dots,e_n)| = 1 + |e_1| + \dots + |e_n|$, where $|A| = 1$ for a variable A, and the *size of* G is $|G| = \sum_{A \in V} |\mathrm{rhs}(A)|$. The size of G in our example is $|G| = |+(1,Y)| + |+(1,1)| = 3 + 3 = 6$.

2.2 String straight-line programs

A *string straight-line program* (SSLP for short) over some alphabet Θ is an SLP over the algebra $(\Theta^*, \{\varepsilon, \circ\} \cup \Theta)$, where ε evaluates to the empty string, $\circ$

is string concatenation, and every symbol from Θ is a constant that evaluates to itself. Instead of $v \circ w$ we often simply write vw. Consider the SSLP $G = (\{A, B, C\}, \mathrm{rhs})$ over $\{a, b\}$ with $\mathrm{rhs}(A) = BB$, $\mathrm{rhs}(B) = CCb$ and $\mathrm{rhs}(C) = aa$, then $[\![A]\!]_G = aaaabaaaab$.

2.3 Forest straight-line programs

We fix an alphabet Σ for the rest of the paper that is used to label nodes in forests. A *forest* is a list of trees $t_1 \dots t_n$ $(n \geq 0)$, while a *tree* is of the form $a(f)$ where $a \in \Sigma$ and f is a forest. Here, a is the root character and its children are the roots of the forest f. The set of forests is denoted by $\mathcal{F}$ and the set of trees by $\mathcal{T}$. *Forests with a hole* are defined as follows: x is a forest with a hole, and $t_1 \dots t_n$ $(n \geq 1)$ is a forest with a hole if $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$ are trees and t_i is a tree with a hole for some $1 \leq i \leq n$. Here, x can be thought of as a placeholder that appears exactly once in a forest with a hole. Trees with a hole are of the form $a(f)$, where $a \in \Sigma$ and f is a forest with a hole. We write $\mathcal{F}_x$ for the set of forests with a hole and $\mathcal{T}_x$ for the set of trees with a hole. The *forest algebra* $\mathcal{A}_\mathcal{F} = (\mathcal{F} \cup \mathcal{F}_x, \mathcal{I}_\mathcal{F})$ has the following operations $\mathcal{I}_\mathcal{F}$:

- $[\![\varepsilon]\!]_\mathcal{F} = \varepsilon$ — the empty forest.
- $[\![x]\!]_\mathcal{F} = x$ — a single hole.
- $[\![fg]\!]_\mathcal{F} = [\![f]\!]_\mathcal{F}[\![g]\!]_\mathcal{F}$ — horizontal concatenation, only defined if $[\![f]\!]_\mathcal{F} \notin \mathcal{F}_x$ or $[\![g]\!]_\mathcal{F} \notin \mathcal{F}_x$.
- $[\![a]\!]_\mathcal{F} = a(x)$ — a single node with a hole.
- $[\![f\langle g\rangle]\!]_\mathcal{F} = [\![f]\!]_\mathcal{F}[[\![g]\!]_\mathcal{F}]$ — substitution, only defined if $[\![f]\!]_\mathcal{F} \in \mathcal{F}_x$.

The notation $f[g]$ means that x in f is replaced with g. The idea of forest algebras goes back to [3].

An SLP $F = (V, \mathrm{rhs})$ over $\mathcal{A}_\mathcal{F}$ is called a *forest straight-line program* or FSLP for short. As a short-hand notation we write $V_0 = \{A \in V \mid [\![A]\!]_F \in \mathcal{F}\}$ for the variables that produce forests without a hole and $V_1 = \{A \in V \mid [\![A]\!]_F \in \mathcal{F}_x\}$ for the variables that produce forests with a hole. *Normal form FSLPs* were introduced in [7], that restrict the rhs-forms that may be used. We are also going to make use of this normal form, since it simplifies the upcoming constructions. First, let $V_0^\perp \subseteq V_0$ be defined as

$$\begin{aligned} V_0^\perp = &\{A \in V_0 \mid \mathrm{rhs}(A) = a\langle C\rangle,\ a \in \Sigma,\ C \in V_0\} \\ &\cup \{A \in V_0 \mid \mathrm{rhs}(A) = B\langle C\rangle,\ B \in V_1,\ C \in V_0\} \end{aligned}$$

For a normal form FSLP we require the following:

- $\mathrm{rhs}(A)$ for every $A \in V_1$ is of the form $a\langle LxR\rangle$, where $a \in \Sigma$, $L, R \in V_0$, or of the form $B\langle C\rangle$, where $B, C \in V_1$.
- $\mathrm{rhs}(A)$ for every $A \in V_0$ is of the form ε, or of the form BC, where $B, C \in V_0$, or of the form $a\langle C\rangle$, where $a \in \Sigma$ and $C \in V_0$, or of the form $B\langle C\rangle$, where $B \in V_1$ and $C \in V_0^\perp$.

Note that $[\![A]\!]_F \in \mathcal{T}$ for all $A \in V_0^{\perp}$ and $[\![B]\!]_F \in \mathcal{T}_x$ for all $B \in V_1$, i.e., only tree contexts appear in a normal form FSLP instead of arbitrary forest contexts.

Lemma 1. *An FSLP $F = (V, \mathrm{rhs})$ can be transformed in linear time into an FSLP F' that is in normal form such that $[\![A]\!]_F = [\![A]\!]_{F'}$ for every $A \in V_0$.*

This was shown in [7]. Since we allow linear time preprocessing, we assume from now on that every FSLP is in normal form.

Example 1. Suppose that $a, b, c \in \Sigma$ and let $n \in \mathbb{N}$. Let $F_n = (V, \mathrm{rhs})$ with

$$V = \{E, A^\ell, A^r\} \cup \{A_i, B_i \mid 0 \le i \le n\} \cup \{C_i^k, D_i^k, G_i^k, H_i^k \mid k \in \{\ell, r\},\ 0 \le i \le n\},$$

and $\mathrm{rhs}(E) = \varepsilon$, $\mathrm{rhs}(A^\ell) = a\langle E\rangle$, $\mathrm{rhs}(A^r) = b\langle E\rangle$, $\mathrm{rhs}(A_0) = A^\ell A^r$, $\mathrm{rhs}(B_0) = b\langle ExE\rangle$,

$$\begin{aligned}
&\mathrm{rhs}(A_{i+1}) = A_i A_i \text{ for } 0 \le i < n,\\
&\mathrm{rhs}(B_{i+1}) = B_i\langle B_i\rangle \text{ for } 0 \le i < n,\\
&\mathrm{rhs}(C_i^\ell) = c\langle A_i x A^r\rangle \text{ for } 0 \le i \le n,\\
&\mathrm{rhs}(C_i^r) = c\langle A^\ell x A_i\rangle \text{ for } 0 \le i \le n,\\
&\mathrm{rhs}(D_i^k) = B_i\langle C_i^k\rangle \text{ for } k \in \{\ell, r\},\ 0 \le i \le n,\\
&\mathrm{rhs}(G_i^k) = C_i^k\langle D_i^k\rangle \text{ for } k \in \{\ell, r\},\ 0 \le i \le n,\\
&\mathrm{rhs}(H_i^k) = G_i^k\langle A^k\rangle \text{ for } k \in \{\ell, r\},\ 0 \le i \le n.
\end{aligned}$$

F_n is in normal form with $V_0^{\perp} = \{A^\ell, A^r\} \cup \{H_i^k \mid 0 \le i \le n,\ k \in \{\ell, r\}\}$. For $0 \le i \le n$ we have $[\![A_i]\!]_{F_n} = (ab)^{2^i}$ and $[\![B_i]\!]_{F_n} = \underbrace{b(\cdots b(}_{2^i} x)\cdots)$,

$$\begin{aligned}
[\![H_i^\ell]\!]_{F_n} &= [\![G_i^\ell\langle A^\ell\rangle]\!]_{F_n} = [\![C_i^\ell\langle D_i^\ell\langle A^\ell\rangle\rangle]\!]_{F_n} = [\![C_i^\ell\langle B_i^\ell\langle C_i^\ell\langle A^\ell\rangle\rangle\rangle]\!]_{F_n}\\
&= c((ab)^{2^i}\,\underbrace{b(\cdots b(}_{2^i}\,c((ab)^{2^i}ab)\,\underbrace{)\cdots)}_{2^i}\,b) \text{ and}\\
[\![H_i^r]\!]_{F_n} &= c(a\,\underbrace{b(\cdots b(}_{2^i}\,c(ab(ab)^{2^i})\,\underbrace{)\cdots)}_{2^i}(ab)^{2^i}).
\end{aligned}$$

Note that the trees $[\![H_n^k]\!]_{F_n}$, where $k \in \{\ell, r\}$, have both exponential width and height in n. See Figure 1 for an illustration.

3 Navigation

The goal of this section is to prove the following theorem:

Theorem 1. *Let $F = (V, \mathrm{rhs})$ be an FSLP in normal form. We can in linear time precompute some data structure of linear size in $|F|$, such that the following operations work in constant time, where $\mathcal{N}(F)$ is the set of node representations, that we will define later:*

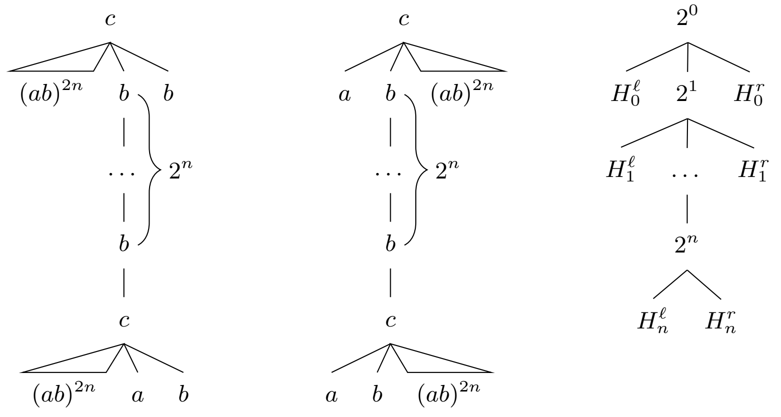

Fig. 1. The trees $[\![H_n^\ell]\!]_{F_n}$ on the left and $[\![H_n^r]\!]_{F_n}$ in the middle from Example 1, and the tree t_M from Example 4 on the right.

- $\mathrm{r}_\triangleleft\colon V_0 \to \mathcal{N}(F) \cup \{\bot\}$: *Return the root of the first tree in the forest represented by an input variable.*
- $\swarrow\colon \mathcal{N}(F) \to \mathcal{N}(F) \cup \{\bot\}$: *Return the first child of the current node.*
- $\to\colon \mathcal{N}(F) \to \mathcal{N}(F) \cup \{\bot\}$: *Return the right sibling of the current node.*
- $\mathrm{r}_\triangleright\colon V_0 \to \mathcal{N}(F) \cup \{\bot\}$: *Return the root of the last tree in the forest represented by an input variable.*
- $\searrow\colon \mathcal{N}(F) \to \mathcal{N}(F) \cup \{\bot\}$: *Return the last child of the current node.*
- $\leftarrow\colon \mathcal{N}(F) \to \mathcal{N}(F) \cup \{\bot\}$: *Return the left sibling of the current node.*
- $\uparrow\colon \mathcal{N}(F) \to \mathcal{N}(F) \cup \{\bot\}$: *Return the parent of the current node.*
- $\mathrm{z}\colon \mathcal{N}(F) \to \Sigma$: *Get the symbol at the current node.*

The special value $\bot$ is used to indicate that an operation may fail. For example, going to the first child of a leaf node returns $\bot$.

The implementation of these operations will make use of SSLP traversals, that can already be done in constant time, which was proven in [12].

Lemma 2. *Let $G = (V, \mathrm{rhs})$ be an SSLP over some alphabet Θ. We can precompute some data structure in linear time in $|G|$, such that the following operations work in constant time, where $\mathcal{N}(G)$ is the set of positions:*

- $\triangleleft\colon V \to \mathcal{N}(G) \cup \{\bot\}$: *Go to the first position in the string derived by a given variable.*
- $\triangleright\colon V \to \mathcal{N}(G) \cup \{\bot\}$: *Go to the last position in the string derived by a given variable.*
- $\mathrm{z}\colon \mathcal{N}(G) \to \Theta$: *Get the symbol at the current position.*
- $\to\colon \mathcal{N}(G) \to \mathcal{N}(G) \cup \{\bot\}$: *Go to the next position.*
- $\leftarrow\colon \mathcal{N}(G) \to \mathcal{N}(G) \cup \{\bot\}$: *Go to the previous position.*

Each element $\gamma \in \mathcal{N}(G)$ represents a position in the string $[\![A]\!]_G$ for a certain variable $A \in V$. We denote this variable A by S_γ.

Like in [12], we first define the *spine SSLP* $F_{\boxbar} = (V_{\boxbar}, \mathrm{rhs}_{\boxbar})$ over $\Sigma_{\boxbar}$ by

$$\begin{aligned}
\Sigma_{\boxbar} &= \{a\langle LxR\rangle \mid \mathrm{rhs}(A) = a\langle LxR\rangle,\ A \in V\} \\
&\quad \cup \{a\langle C\rangle \mid \mathrm{rhs}(A) = a\langle C\rangle,\ A \in V\}, \\
V_{\boxbar} &= V_1 \cup V_0^{\perp}, \\
\mathrm{rhs}_{\boxbar}(A) &= \begin{cases} BC & \text{if } \mathrm{rhs}(A) = B\langle C\rangle,\ A \in V_1, \\ a\langle LxR\rangle & \text{if } \mathrm{rhs}(A) = a\langle LxR\rangle, \\ a\langle C\rangle & \text{if } \mathrm{rhs}(A) = a\langle C\rangle, \\ \mathrm{rhs}(B) & \text{if } \mathrm{rhs}(A) = B\langle C\rangle,\ A \in V_0^{\perp}. \end{cases}
\end{aligned}$$

The idea of the spine SSLP is that its symbols $\Sigma_{\boxbar}$ are exactly the rhs-expressions of F where symbols from Σ occur. Navigating to one of the symbols in $[\![A]\!]_{F_{\boxbar}}$ for $A \in V_0^{\perp}$ is essentially the same as navigating to a specific node in $[\![A]\!]_F$. If $\mathrm{rhs}(A) = a\langle C\rangle$ this string only consists of a single symbol $a\langle C\rangle$. If instead $\mathrm{rhs}(A) = B\langle C\rangle$, the word $[\![A]\!]_{F_{\boxbar}}$ is of the form $a_1\langle L_1xR_1\rangle \dots a_m\langle L_mxR_m\rangle$. Navigating such a word left or right corresponds to navigating up or down in the tree $[\![A]\!]_F$, while following the x position. Standing on the symbol $a_i\langle L_ixR_i\rangle$ means that the current node is labelled with a_i. When we walk past the last element, $a_m\langle L_mxR_m\rangle$, we have to continue to navigate in $[\![C]\!]_F$. Let us thus define the vertical navigation structure as $\mathcal{V}(F) = \mathcal{N}(F_{\boxbar})^+$, which enables us to chain multiple navigations in $F_{\boxbar}$ together. We can then define the following navigation steps, which we will later use to implement the actual navigation.

- $\triangle\colon V_0^{\perp} \to \mathcal{V}(F)$: Go to the root node,
- $\uparrow\colon \mathcal{V}(F) \to \mathcal{V}(F) \cup \{\perp\}$: Go to the parent node.
- $\downarrow\colon \mathcal{V}(F) \to \mathcal{V}(F) \cup \{\perp\}$: Go to the child node at the x position.
- $\mathrm{z}\colon \mathcal{V}(F) \to \Sigma_{\boxbar}$: Get the current symbol.

Implementing these is straight-forward: We set $\triangle(A) = \triangleleft(A)$ for $A \in V_0^{\perp}$. Let $v \in \mathcal{N}(F_{\boxbar})^*$ and $\gamma \in \mathcal{N}(F_{\boxbar})$. We set $\mathrm{z}(v\gamma) = \mathrm{z}(\gamma)$. The other two operations are defined as follows:

$$\uparrow(v\gamma) = \begin{cases} v \leftarrow(\gamma) & \text{if } \leftarrow(\gamma) \neq \perp, \\ v & \text{if } \leftarrow(\gamma) = \perp \text{ and } v \neq \varepsilon, \\ \perp & \text{if } \leftarrow(\gamma) = \perp \text{ and } v = \varepsilon, \end{cases}$$

$$\downarrow(v\gamma) = \begin{cases} v \rightarrow(\gamma) & \text{if } \rightarrow(\gamma) \neq \perp, \\ v\gamma \triangleleft(C) & \text{if } \rightarrow(\gamma) = \perp \text{ and } \mathrm{rhs}(S_\gamma) = B\langle C\rangle, \\ \perp & \text{if } \rightarrow(\gamma) = \perp \text{ and } \mathrm{rhs}(S_\gamma) = a\langle C\rangle. \end{cases}$$

Defining the operations on $\mathcal{V}(F)$ in isolation not only makes the definition of the actual navigation easier, but it also provides the benefit that we only have to change these when we add the ability to do subtree equality checks, while the definition of the actual navigation will stay the same.

What is left to do is to add the ability to do horizontal navigations as well. Horizontal navigations can happen on any of the L and R in $a\langle LxR\rangle$ as well as C in $a\langle C\rangle$. For this, we define another auxiliary SSLP, called the *rib SSLP* $F_{\boxminus} = (V_{\boxminus}, \mathrm{rhs}_{\boxminus})$ over $\Sigma_{\boxminus}$ by $\Sigma_{\boxminus} = \{\underline{A} \mid A \in V_0^{\perp}\}$, $V_{\boxminus} = V_0$ and

$$\mathrm{rhs}_{\boxminus}(A) = \begin{cases} \varepsilon & \text{if } \mathrm{rhs}(A) = \varepsilon, \\ BC & \text{if } \mathrm{rhs}(A) = BC, \\ \underline{A} & \text{if } A \in V_0^{\perp}. \end{cases}$$

We make a copy of all $A \in V_0$ which we simply call $\underline{A}$. This is because we actually want an SSLP navigation on $\Sigma_{\boxminus}$ to end in a symbol from $V_0^{\perp}$. Without making the copy, we would not be able to assign an rhs value to such a symbol. Note that for each $A \in V_0$ we have $\llbracket A \rrbracket_{F_{\boxminus}} = \underline{A_1} \ldots \underline{A_n}$, where $A_i \in V_0^{\perp}$. Thus $\llbracket A_i \rrbracket_F$ is the ith tree in the forest $\llbracket A \rrbracket_F$, i.e. $\llbracket A \rrbracket_F = \llbracket A_1 \rrbracket_F \cdots \llbracket A_n \rrbracket_F \in \mathcal{T}^n$.

Example 2. Using F_n from Example 1, we have

$$\begin{aligned} \Sigma_{\boxbar} &= \{b\langle ExE\rangle, a\langle E\rangle, b\langle E\rangle\} \cup \{c\langle A_i x A^r\rangle, c\langle A^\ell x A_i\rangle \mid 0 \le i \le n\}, \\ \Sigma_{\boxminus} &= \{\underline{A^\ell}, \underline{A^r}\} \cup \{\underline{H_i^k} \mid 0 \le i \le n,\ k \in \{\ell, r\}\} \text{ and for example} \\ \llbracket H_i^\ell \rrbracket_{F_{\boxbar}} &= c\langle A_i x A^r\rangle (b\langle ExE\rangle)^{2^i} c\langle A_i x A^r\rangle \text{ for } 0 \le i \le n \text{ and} \\ \llbracket A_i \rrbracket_{F_{\boxminus}} &= (\underline{A^\ell}\,\underline{A^r})^{2^i} \text{ for } 0 \le i \le n. \end{aligned}$$

For horizontal navigations, we have to remember if we started in an L or R in $a\langle LxR\rangle$ or in C in $a\langle C\rangle$, which we record as ℓ, r and m, respectively. Therefore, the horizontal navigation structure is $\mathcal{H}(F) = \{\ell, m, r\} \times \mathcal{N}(F_{\boxminus})$. The idea for the whole navigation is then to interleave navigations on $F_{\boxminus}$ with navigations on $F_{\boxbar}$, so we define $\mathcal{N}(F) = (\mathcal{H}(F) \times \mathcal{V}(F))^+$.

We first introduce a short-hand notation to create a navigation on $F_{\boxminus}$ followed by a navigation on $F_{\boxbar}$. Let $\triangleleft_d \colon V_0 \to (\mathcal{H}(F) \times \mathcal{V}(F)) \cup \{\perp\}$ for every $d \in \{\ell, m, r\}$ be defined by

$$\triangleleft_d(A) = \begin{cases} ((d, \triangleleft(A)), \triangle(D)) & \text{if } \triangleleft(A) \neq \perp \text{ and } \mathrm{z}(\triangleleft(A)) = \underline{D}, \\ \perp & \text{if } \triangleleft(A) = \perp. \end{cases}$$

We are now going to implement the operations from Theorem 1. Suppose that the current state is $w((d,h),v) \in \mathcal{N}(F)$, where $w \in (\mathcal{H}(F) \times \mathcal{V}(F))^*$, $(d,h) \in \mathcal{H}(F)$, so $d \in \{\ell, m, r\}$ and $h \in \mathcal{N}(F_{\boxminus})$, and $v \in \mathcal{V}(F)$. To query the current symbol, we define $\mathrm{z}(w((d,h),v)) = a$ if $\mathrm{z}(v) = a\langle LxR\rangle$ or $\mathrm{z}(v) = a\langle C\rangle$. To implement $\swarrow$, we have to consider the following cases: If we are on a symbol of the form $\mathrm{z}(v) = a\langle LxR\rangle$ we can either have $\llbracket L \rrbracket_F \neq \varepsilon$ or $\llbracket L \rrbracket_F = \varepsilon$. In the first case we go to the first symbol of $\llbracket L \rrbracket_{F_{\boxminus}}$ and record ℓ. This symbol is of the form $\underline{A}$, where $A \in V_0^{\perp}$, so we have to go to the root node of $\llbracket A \rrbracket_F$. We therefore append $\triangleleft_\ell(L) = ((\ell, \triangleleft(L)), \triangle(A))$. In case $\llbracket L \rrbracket_F = \varepsilon$ we reach the x of $a\langle LxR\rangle$, which means that we have to move the current navigation on $\mathcal{V}(F)$ down one position, i.e. we replace v with $\downarrow(v)$. If we are on a symbol of the form $a\langle C\rangle$ we can again have that $\llbracket C \rrbracket_F = \varepsilon$, in which case there is nowhere to go. If $\llbracket C \rrbracket_F \neq \varepsilon$ we go to the first symbol $\underline{A}$ of $[\![C]\!]_{F_\boxminus}$, record m and again start a navigation to the root of $[\![A]\!]_{F_\boxminus}$, so we append $\triangleleft_m(C) = ((m, \triangleleft(C)), \triangle(A))$. Altogether, the function is

$$\swarrow(w((d,h),v)) = \begin{cases} w((d,h),v)\triangleleft_\ell(L) & \text{if } \mathrm{z}(v) = a\langle LxR\rangle \text{ and } \triangleleft_\ell(L) \neq \bot, \\ w((d,h),\downarrow(v)) & \text{if } \mathrm{z}(v) = a\langle LxR\rangle \text{ and } \triangleleft_\ell(L) = \bot, \\ w((d,h),v)\triangleleft_m(C) & \text{if } \mathrm{z}(v) = a\langle C\rangle \text{ and } \triangleleft_m(C) \neq \bot, \\ \bot & \text{if } \mathrm{z}(v) = a\langle C\rangle \text{ and } \triangleleft_m(C) = \bot, \end{cases}$$

The function $\searrow$ is symmetric and is therefore omitted. Going to the left or right neighbor is more involved. Since going to the right neighbor is basically a mirrored version of going to the left neighbor, we will focus on the former. We start by trying to move the current navigation on $\mathcal{V}(F)$ one position up, i.e. replace v with $\uparrow(v)$, because we need to know what is to the right of the current node. If moving up succeeds, the current symbol is of the form $a\langle LxR\rangle$, which means that we were standing on the x position and thus the next tree we have to go to is the first tree of $[\![R]\!]_F$ if $[\![R]\!]_F \neq \varepsilon$. We then go to the first symbol $\underline{A}$ of $[\![R]\!]_{F_\boxminus}$, record r, and go to the root node of $[\![A]\!]_F$, which means we append $\triangleleft_r(R) = ((r, \triangleleft(R)), \triangle(A))$. If $[\![R]\!]_F = \varepsilon$, there is nowhere to go. If moving up does not succeed, it means that v points to the root node. We are therefore on an $\underline{A_i}$ of the previous horizontal navigation h, which is of the form $\underline{A_1}\ldots\underline{A_n}$. If $i < n$, then we move this navigation one to the right and go to the root node of $[\![A_{i+1}]\!]_F$, so we replace $((d,h),v)$ with $((d,\rightarrow(h)),\triangle(A_{i+1}))$. If $i = n$, then we have to look at the current symbol of the last navigation from w. In case $w = \varepsilon$, there is nowhere to go. Now suppose w ends in $v' \in \mathcal{V}(F)$. Suppose that the current symbol of v' is of the form $\mathrm{z}(v') = a\langle LxR\rangle$ and $d = \ell$. This means that we left the navigation on L to the right and end up on x, so we have to move the vertical navigation one position down, i.e. we replace v' with $\downarrow(v')$. In case $d = r$, we left the navigation on R to the right, so there is nowhere to go. If the current symbol is instead of the form $\mathrm{z}(v') = a\langle C\rangle$ there is also nowhere to go, since we were on the last tree of $[\![C]\!]_F$. Altogether, we have the function

$$\rightarrow(w((d,h),v)) = \begin{cases} w((d,h),u)\triangleleft_r(R) & \text{if } \uparrow(v) = u,\ \mathrm{z}(u) = a\langle LxR\rangle \\ & \text{and } \triangleleft_r(R) \neq \bot, \\ \bot & \text{if } \uparrow(v) = u,\ \mathrm{z}(u) = a\langle LxR\rangle \\ & \text{and } \triangleleft_r(R) = \bot, \\ w((d,h'),\triangle(D)) & \text{if } \uparrow(v) = \bot,\ \rightarrow(h) = h' \neq \bot \\ & \text{and } \mathrm{z}(h') = \underline{D}, \\ w'((d',h'),\downarrow(v')) & \text{if } \uparrow(v) = \bot,\ \rightarrow(h) = \bot, \\ & w = w'((d',h'),v'), \\ & w' \in (\mathcal{H}(F) \times \mathcal{V}(F))^*,\ (d',h') \in \mathcal{H}(F), \\ & v' \in \mathcal{V}(F) \text{ and } d = \ell, \\ \bot & \text{if } \uparrow(v) = \bot,\ \rightarrow(h) = \bot,\ w = \varepsilon \\ & \text{or } d \neq \ell. \end{cases}$$

Going to the parent node is straight-forward. We try to replace v with $\uparrow(v)$. If this is not possible, we remove (d,h) and v, if $w \neq \varepsilon$. If $w = \varepsilon$, then there is nowhere to go. Formally, this is

$$\uparrow(w(((d,h),v)) = \begin{cases} w((d,h),\uparrow(v)) & \text{if } \uparrow(v) \neq \bot, \\ w & \text{if } \uparrow(v) = \bot \text{ and } w \neq \varepsilon, \\ \bot & \text{if } \uparrow(v) = \bot \text{ and } w = \varepsilon, \end{cases}$$

For starting a navigation in $S \in V_0$, we go to the first tree of $[\![S]\!]_F$ and record m in case $[\![S]\!]_F \neq \varepsilon$. This navigation ends on a symbol $\underline{A}$, and we go to the root node of $[\![A]\!]_F$. Thus we start with $\mathrm{r}_\triangleleft(S) = \triangleleft_m(S) = ((m,\triangleleft(S)),\triangle(A))$. If $[\![S]\!]_F = \varepsilon$ there is nowhere to go, so $\mathrm{r}_\triangleleft(S) = \bot$. Going to the last tree is symmetric.

Example 3. Consider F_n from Example 1. We have

$$\mathrm{r}_\triangleleft(H_n^\ell) = ((m,\triangleleft(H_n^\ell)),\triangle(H_n^\ell))$$

and since $\mathrm{z}(\triangle(H_n^\ell)) = c\langle A_n x A^r\rangle$, we obtain $\mathrm{z}(\mathrm{r}_\triangleleft(H_n^\ell)) = c$. Now consider applying $\swarrow$ to this node: Since $\mathrm{z}(\triangleleft(A_n)) = \underline{A}^\ell$, we have

$$\swarrow(\mathrm{r}_\triangleleft(H_n^\ell)) = ((m,\triangleleft(H_n^\ell)),\triangle(H_n^\ell))((\ell,\triangleleft(A_n)),\triangle(A^\ell)).$$

Using z on this structure yields a, because $\mathrm{z}(\triangle(A^\ell)) = a\langle E\rangle$.

4 Navigation with Equality Checks

In this section we change our navigation structure, which we again call $\mathcal{N}(F)$, to include subtree equality checks.

Theorem 2. *Using polynomial time preprocessing, we can precompute some data structure of linear size in $|F|$ such that in addition to the operations from Theorem 1 the following operation, which checks if two subtrees rooted at the given input nodes are equal:*

$$\mathrm{eq}\colon \mathcal{N}(F) \times \mathcal{N}(F) \to \{0,1\}.$$

We ensure that our FSLP is *reduced* which means that there are no $A \neq A' \in V$ such that $[\![A]\!]_F = [\![A']\!]_F$, which can be tested using a result from [7]. We give a similar characterization of equal subtrees as the one found in [12]. Let us write $A_\boxminus$ instead of $[\![A]\!]_{F_\boxminus}$. We define $V_0^\boxminus = \{A \in V_0^\perp \mid \mathrm{rhs}(A) = B\langle C\rangle\}$. For $A \in V_0^\boxminus$ with $\mathrm{rhs}(A) = B\langle C\rangle$ let the *i'th subtree* be defined as $A_\triangle(i) = [\![B_\boxminus[i]\langle \cdots B_\boxminus[\ell(A)]\langle C\rangle \cdots\rangle]\!]_F$, where $1 \leq i \leq \ell(A)+1$ and $\ell(A) = |B_\boxminus|$. Note that $A_\triangle(1) = [\![A]\!]_F$ and $A_\triangle(\ell(A)+1) = [\![C]\!]_F$. Now let $i \geq 2$ be the smallest number such that there is a $D \in V_0^\perp$ with $A_\triangle(i) = [\![D]\!]_F$. We call i the *split index* of A, written as $\mathrm{si}(A)$, and D the *split variable* of A, written as $\mathrm{sv}(A)$. Since $A_\triangle(\ell(A)+1) = [\![C]\!]_F$ and $C \in V_0^\perp$, the split index and split variable always exist. The idea to implement the navigation that also supports subtree equality checks is to always stay below the split index. When we reach the split index, we simply continue to navigate in the split variable, which preserves subtree equality. We now only have to characterize the equal subtrees below split indices.

Lemma 3. *Let $t, t' \in \mathcal{T}$, $a, a' \in \Sigma$ and $L, L', R, R' \in V_0$, with*

1. $a\langle LxR\rangle \neq a'\langle L'xR'\rangle$, *and*
2. $[\![a\langle LxR\rangle]\!]_F[t] = [\![a'\langle L'xR'\rangle]\!]_F[t']$.

Then there are $D, D' \in V_0^{\perp}$ with $[\![D]\!]_F = t'$ and $[\![D']\!]_F = t$.

Proof. Since F is in normal form, there are variables

$$\{L_1, \ldots, L_n, R_1, \ldots, R_m, L'_1, \ldots, L'_{n'}, R'_1, \ldots, R'_{m'}\} \subseteq V_0^{\perp}$$

with $[\![L]\!]_F = [\![L_1 \ldots L_n]\!]_F$, $[\![R]\!]_F = [\![R_1 \ldots R_m]\!]_F$, $[\![L']\!]_F = [\![L'_1 \ldots L'_{n'}]\!]_F$ and $[\![R']\!]_F = [\![R'_1 \ldots R'_{m'}]\!]_F$. From Point 2 we obtain $a = a'$ and $[\![L]\!]_F\, t\, [\![R]\!]_F = [\![L']\!]_F\, t'\, [\![R']\!]_F$. From $[\![L]\!]_F = [\![L']\!]_F$ we would obtain $t = t'$ and $[\![R]\!]_F = [\![R']\!]_F$ which is in contradiction to Point 1 because F is reduced. Hence we must have $[\![L]\!]_F \neq [\![L']\!]_F$ which implies that $[\![L_1 \ldots L_n]\!]_F \in \mathcal{T}^n$ is a proper prefix of $[\![L'_1 \ldots L'_{n'}]\!]_F \in \mathcal{T}^{n'}$ or vice versa. In the first case we have $t = [\![L'_{n+1}]\!]_F$ and $t' = [\![R_{n'-n}]\!]_F$, in the second case $t' = [\![L_{n'+1}]\!]_F$ and $t = [\![R'_{n-n'}]\!]_F$.

Lemma 4. *Let $A, A' \in V_0^{\boxbar}$, $1 \leq i < \mathrm{si}(A)$ and $1 \leq i' < \mathrm{si}(A')$. We have $A_\triangle(i) = A'_\triangle(i')$ if and only if*

1. $A_{\boxbar}[i : \mathrm{si}(A) - 2] = A'_{\boxbar}[i' : \mathrm{si}(A') - 2]$, *and*
2. $[\![A_{\boxbar}[\mathrm{si}(A) - 1]\langle \mathrm{sv}(A)\rangle]\!]_F = [\![A'_{\boxbar}[\mathrm{si}(A') - 1]\langle \mathrm{sv}(A')\rangle]\!]_F$.

Proof. It is easy to see that Points 1 and 2 imply $A_\triangle(i) = A'_\triangle(i')$. To prove the opposite direction, we use induction on $m = \min\{\mathrm{si}(A) - i - 1,\ \mathrm{si}(A') - i' - 1\}$. Assume that $A_\triangle(i) = A'_\triangle(i')$. Let $m = 0$, which means that either $i = \mathrm{si}(A) - 1$ or $i' = \mathrm{si}(A') - 1$. We assume that $i = \mathrm{si}(A) - 1$ and show that $i' = \mathrm{si}(A') - 1$. By definition of si and sv we have $[\![A_{\boxbar}[\mathrm{si}(A) - 1]\langle \mathrm{sv}(A)\rangle]\!]_F = A_\triangle(i)$. Since $A'_\triangle(i') = [\![A'_{\boxbar}[i']]\!]_F[A'_\triangle(i' + 1)]$, we obtain

$$[\![A_{\boxbar}[\mathrm{si}(A) - 1]\langle \mathrm{sv}(A)\rangle]\!]_F = [\![A'_{\boxbar}[i']]\!]_F[A'_\triangle(i' + 1)].$$

If $A_{\boxbar}[\mathrm{si}(A) - 1] = A'_{\boxbar}[i']$ we have $[\![\mathrm{sv}(A)]\!]_F = A'_\triangle(i' + 1)$. If $A_{\boxbar}[\mathrm{si}(A) - 1] \neq A'_{\boxbar}[i']$, then we obtain from Lemma 3 that there is a $D \in V_0^{\perp}$ with $[\![D]\!]_F = A'_\triangle(i' + 1)$. In both cases, there is a variable that evaluates to $A'_\triangle(i' + 1)$, and since $i' < \mathrm{si}(A')$, we must have that $i' + 1 = \mathrm{si}(A')$ and $A'_\triangle(i' + 1) = [\![\mathrm{sv}(A')]\!]_F$. Therefore, we obtain that $A_{\boxbar}[i : \mathrm{si}(A) - 2] = \varepsilon$, $A'_{\boxbar}[i' : \mathrm{si}(A') - 2] = \varepsilon$ and

$$[\![A_{\boxbar}[\mathrm{si}(A) - 1]\langle \mathrm{sv}(A)\rangle]\!]_F = [\![A'_{\boxbar}[\mathrm{si}(A') - 1]\langle \mathrm{sv}(A')\rangle]\!]_F.$$

The symmetric case, in which $i' = \mathrm{si}(A') - 1$, uses the same arguments. Now let $m > 0$, so $i < \mathrm{si}(A) - 1$ and $i' < \mathrm{si}(A') - 1$. Since $A_\triangle(i) = A'_\triangle(i')$ we obtain that $[\![A_{\boxbar}[i]]\!]_F[A_\triangle(i + 1)] = [\![A'_{\boxbar}[i']]\!]_F[A'_\triangle(i' + 1)]$. If $A_{\boxbar}[i] \neq A'_{\boxbar}[i']$ we would obtain from Lemma 3 that there are $D, D' \in V_0^{\perp}$ with $[\![D]\!]_F = A'_\triangle(i' + 1)$ and $[\![D']\!]_F = A_\triangle(i + 1)$ which contradicts $i < \mathrm{si}(A) - 1$ as well as $i' < \mathrm{si}(A') - 1$. Therefore, we must have $A_{\boxbar}[i] = A'_{\boxbar}[i']$. From $A_\triangle(i) = A'_\triangle(i')$ and this fact we can conclude that $A_\triangle(i + 1) = A'_\triangle(i' + 1)$. Therefore by induction we have $A_{\boxbar}[i + 1 : \mathrm{si}(A) - 2] = A'_{\boxbar}[i' + 1 : \mathrm{si}(A') - 2]$, as well as Point 2. Together with $A_{\boxbar}[i] = A'_{\boxbar}[i']$ we also obtain Point 1.

To use Lemma 4 for equality checks, we still have to argue that we can implement some data structure of linear size that allows to perform these checks in constant time. To check Point 2 of Lemma 4 we do the following: Let

$$\sim = \{(A, A') \in V_0^{\boxbar} \times V_0^{\boxbar} \mid [\![A_{\boxbar}[\mathrm{si}(A)-1]\langle \mathrm{sv}(A)\rangle]\!]_F = [\![A'_{\boxbar}[\mathrm{si}(A')-1]\langle \mathrm{sv}(A')\rangle]\!]_F\}.$$

This relation is an equivalence relation. We assign each equivalence class a natural number and precompute a mapping that takes an element to its equivalence class, represented as this number. This mapping requires linear space and we can test if two elements belong to the same equivalence class in constant time. Let $A, A' \in V_0^{\boxbar}$, $1 \leq i < \mathrm{si}(A)$ and $1 \leq i' < \mathrm{si}(A')$. We only have to check Point 1 of Lemma 4 if Point 2 is true, so suppose that $(A, A') \in \sim$. We now have to test if $A_{\boxbar}[i : \mathrm{si}(A)-2] = A'_{\boxbar}[i' : \mathrm{si}(A')-2]$. This can only be true if $k := \mathrm{si}(A) - 2 - i = \mathrm{si}(A') - 2 - i'$. Let $\mathrm{suff}(A, A')$ be the length of the longest common suffix of $A_{\boxbar}[: \mathrm{si}(A) - 2]$ and $A'_{\boxbar}[: \mathrm{si}(A') - 2]$, which can be computed in polynomial time using a result from [14]. We then have $A_{\boxbar}[i : \mathrm{si}(A) - 2] = A'_{\boxbar}[i' : \mathrm{si}(A') - 2]$ if and only if $k \leq \mathrm{suff}(A, A')$. Storing suff explicitly for all elements belonging to the same equivalence class $M \in V_0^{\boxbar}/\sim$ requires quadratic space. Instead, we compute for each M a tree t_M that has linear size in $|M|$, which we can use to query suff in constant time. In this tree, the elements from M are the leaves and the *lowest common ancestor* of two leaves $A \neq A' \in M$ is labelled with $\mathrm{suff}(A, A')$. Lowest common ancestor queries can be performed in constant time after linear time preprocessing, using the result from [15].

The trees t_M can be constructed as follows: We start with any $A \neq A' \in M$ and make A and A' children of a node labelled with $\mathrm{suff}(A, A')$. Now suppose we have constructed a tree for some elements of M. To add a new $A \in M$ to the tree, we take a leaf A' where $\mathrm{suff}(A, A')$ is maximal. We then find the closest ancestor node a of A' whose parent p is labelled with $m \leq \mathrm{suff}(A, A')$, or in case this does not exist then a is the root node. If $m = \mathrm{suff}(A, A')$ then A becomes a new child of a. If $m < \mathrm{suff}(A, A')$ then we add a new node between a and p, label it with $\mathrm{suff}(A, A')$ and add A as its second child. If a is the root node then we add a new parent to a, label it with $\mathrm{suff}(A, A')$ and add A as its second child.

It remains to argue why we can precompute si and sv in polynomial time, which we can do as follows: For every $A \in V_0^{\boxbar}$ and $A' \neq A \in V_0^{\perp}$ we test if there is an $1 < i \leq \ell(A)$ with $A_{\triangle}(i) = [\![A']\!]_F$, which is done as follows: Since $|A_{\triangle}(1)| > \cdots > |A_{\triangle}(\ell(A))|$ we can use binary search to test if there is an $1 < i \leq \ell(A)$ such that $|A_{\triangle}(i)| = |[\![A']\!]_F|$. For a given i computing $|A_{\triangle}(i)|$ can be done in polynomial time because we can compute an FSLP G with variable X such that $[\![X]\!]_G = A_{\triangle}(i)$. This is done by removing a prefix of $A_{\boxbar}$, which can be done by cutting the syntax tree of A. Also, given a variable X of an FSLP G it is easy to compute $|[\![X]\!]_G|$. Furthermore, we can test if $A_{\triangle}(i) = [\![A']\!]_F$ because given two variables X, Y of an FSLP G we can test in polynomial time if $[\![X]\!]_G = [\![Y]\!]_G$ using a result from [7]. For a given $A \in V_0^{\boxbar}$ we then take the $A' \in V_0^{\perp}$ with the smallest i such that $A_{\triangle}(i) = [\![A']\!]_F$ and set $\mathrm{sv}(A) = A'$ and $\mathrm{si}(A) = i$. If no such i exists then we set $\mathrm{sv}(A) = C$, where $\mathrm{rhs}(A) = B\langle C\rangle$, and $\mathrm{si}(A) = \ell(A) + 1$.

Example 4. Recall the definition of F_n from Example 1. Since $[\![C_i^\ell\langle A^\ell\rangle]\!]_{F_n} = c((ab)^{2^i}xb)[a] = c(ax(ab)^{2^i})[b] = [\![C_i^\ell\langle A^\ell\rangle]\!]_{F_n}$, we have $H^\ell_{i\,\triangle}(2^i+2) = H^r_{i\,\triangle}(2^i+2)$ for $0 \le i \le n$. We also have $H^\ell_{i\,\boxbar}[j] = b\langle ExE\rangle = H^r_{i\,\boxbar}[j]$ for all $2 \le j \le 2^i+1$. This implies that $H^\ell_{i\,\triangle}(j) = H^r_{i\,\triangle}(j)$ for all $2 \le j \le 2^i+1$. Therefore, $\mathrm{si}(H_i^\ell) = \mathrm{si}(H_i^r) = 2^i+3$ and $\mathrm{sv}(H_i^k) = A^k$ for all $0 \le i \le n$ and $k \in \{\ell, r\}$. Since

$$[\![H^\ell_{i\,\boxbar}[\mathrm{si}(H_i^\ell)-1]\langle \mathrm{sv}(H_i^\ell)\rangle]\!]_{F_n} = [\![H^r_{i\,\boxbar}[\mathrm{si}(H_i^r)-1]\langle \mathrm{sv}(H_i^r)\rangle]\!]_{F_n}$$

we have $H_i^\ell \sim H_i^r$ for all $0 \le i \le n$, thus $V_0^\perp/{\sim} = \{\{H_i^\ell, H_i^r\} \mid 0 \le i \le n\}$, and $\mathrm{suff}(H_i^\ell, H_i^r) = 2^i$.

Let us change the definition of $\mathrm{rhs}(D_i^k)$ from $\mathrm{rhs}(D_i^k) = B_i\langle C_i^k\rangle$ to $\mathrm{rhs}(D_i^k) = B_i\langle C_0^k\rangle$. We then have for $0 \le i \le n$ that

$$\begin{aligned}
[\![H_i^\ell]\!]_{F_n} &= c((ab)^{2^i}\underbrace{b(\cdots b(}_{2^i}c(abab)\underbrace{)\cdots)}_{2^i}b) \text{ and}\\
[\![H_i^r]\!]_{F_n} &= c(a\underbrace{b(\cdots b(}_{2^i}c(abab)\underbrace{)\cdots)}_{2^i}(ab)^{2^i}).
\end{aligned}$$

Since $[\![C_0^r\langle A^r\rangle]\!]_{F_n} = c(axab)[b] = c(abxb)[a] = [\![C_0^\ell\langle A^\ell\rangle]\!]_{F_n}$, we have

$$H^\ell_{i\,\triangle}(2^i+2) = H^r_{i\,\triangle}(2^i+2) = H^\ell_{j\,\triangle}(2^j+2) = H^r_{j\,\triangle}(2^j+2)$$

for all $0 \le i, j \le n$. Thus $H^k_{i\,\triangle}(2^i+2-m) = H^{k'}_{j\,\triangle}(2^j+2-m)$ for all $k, k' \in \{\ell, r\}$, $1 \le i, j \le 2^n$ and $0 \le m \le \max\{2^i, 2^j\}$. Therefore, $V_0^\perp/{\sim} = \{M\}$ consists of the single equivalence class $M = \{H_i^k \mid 0 \le i \le n,\ k \in \{\ell, r\}\}$. See Figure 1 for the tree t_M.

We now explain how we have to change our vertical navigation structure $\mathcal{V}(F)$ and the operations on it to support the subtree equality check eq. We change the $\mathcal{V}(F)$-part of our navigation structure to $\mathcal{V}(F) = (\mathcal{N}(F_\boxbar) \times \mathbb{N})^+$, where we use the $\mathbb{N}$ component to count how many $\downarrow$ steps we made. The operations $\triangle$, $\uparrow$ and z are straight-forward to implement. Let $v \in (\mathcal{N}(F_\boxbar) \times \mathbb{N})^*$, $\gamma \in \mathcal{N}(F_\boxbar)$ and $i \in \mathbb{N}$. We set $\mathrm{z}(v(\gamma, i)) = \mathrm{z}(\gamma)$ and $\triangle(A) = (\triangleleft(A), 1)$ for $A \in V_0^\perp$ and

$$\uparrow(v(\gamma, i)) = \begin{cases} v(\leftarrow(\gamma), i-1) & \text{if } \leftarrow(\gamma) \neq \perp,\\ v & \text{if } \leftarrow(\gamma) = \perp \text{ and } v \neq \varepsilon,\\ \perp & \text{if } \leftarrow(\gamma) = \perp \text{ and } v = \varepsilon, \end{cases}$$

For the implementation of $\downarrow$, let $v(\gamma, i) \in \mathcal{V}(F)$ be the current state, where $v \in (\mathcal{N}(F_\boxbar) \times \mathbb{N})^*$, $\gamma \in \mathcal{N}(F_\boxbar)$ and $i \in \mathbb{N}$. Suppose the navigation γ started in $\triangleleft(A)$. This means that we are currently on $A_\boxbar[i]$ and want to navigate to $A_\boxbar[i+1]$. If $A \notin V_0^\boxbar$, so $\mathrm{rhs}(A)$ is of the form $a\langle C\rangle$, there is nowhere to go. Now let $A \in V_0^\boxbar$. In case $\mathrm{si}(A) > i+1$, we can stay on $A_\boxbar$, so we replace γ with $\rightarrow(\gamma)$ and i with $i+1$. In case $\mathrm{si}(A) = i+1$, we have to continue to navigate in $\mathrm{sv}(A)_\boxbar$, since $A_\triangle(i+1) = [\![\mathrm{sv}(A)]\!]_F$. Therefore, we append $(\triangleleft(\mathrm{sv}(A)), 1)$. Formally, we have

$$\downarrow(v(\gamma, i)) = \begin{cases} v(\rightarrow(\gamma), i+1) & \text{if } S_\gamma \in V_0^{\boxbar} \text{ and } \mathrm{si}(S_\gamma) > i+1, \\ v(\gamma, i) \triangle(\mathrm{sv}(S_\gamma)) & \text{if } S_\gamma \in V_0^{\boxbar} \text{ and } \mathrm{si}(S_\gamma) = i+1, \\ \perp & \text{if } S_\gamma \notin V_0^{\boxbar}. \end{cases}$$

With the new definition of $\mathcal{N}(F)$, the subtree equality check eq can easily be implemented. Suppose the rightmost elements from $\mathcal{N}(F)$ are $v(\gamma, i)$ and $v'(\gamma', i')$, where γ started with $\triangleleft(A)$ and γ' with $\triangleleft(A')$ for $A, A' \in V_0^\perp$. In case $\mathrm{rhs}(A) = a\langle C\rangle$, so $A \in V_0^\perp \setminus V_0^{\boxbar}$, then $A_\triangle(i) = A'_\triangle(i')$ if and only if $\mathrm{rhs}(A') = a\langle C\rangle$. Now let $A, A' \in V_0^{\boxbar}$, in which case we use Lemma 4 to test whether $A_\triangle(i) = A'_\triangle(i')$.

5 Discussion

We first implemented a data structure that can be precomputed in linear time with which we can do navigation steps in constant time. Later we added the ability to do subtree equality checks, again by precomputing a data structure of linear size. However, the precomputation time required is polynomial in this case. It would be interesting to show a lower bound for the exponent. Since the preprocessing requires equality checks for which the best known algorithm is quadratic (see [10]), it would be surprising if this exponent was lower than 2. Implementing all the algorithms of this work would also be interesting. In [6] it was shown that using linear time we can transform an FSLP into an equivalent one such that the height $h(F)$ of its syntax tree is logarithmic in $|F|$. If we can show that $h(F)$ does not increase when using Lemma 1, and that the size $|X|$ of elements $X \in \mathcal{N}(F)$ is bounded by $h(F)$, then we would obtain that $|X| \in \mathcal{O}(\log |F|)$.

References

1. Bille, P., Gørtz, I.L., Landau, G.M., Weimann, O.: Tree compression with top trees. Inf. Comput. **243**, 166–177 (2015). https://doi.org/10.1016/j.ic.2014.12.012, `https://doi.org/10.1016/j.ic.2014.12.012`
2. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM J. Comput. **44**(3), 513–539 (2015). https://doi.org/10.1137/130936889, `https://doi.org/10.1137/130936889`
3. Bojanczyk, M., Walukiewicz, I.: Forest algebras. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]. Texts in Logic and Games, vol. 2, pp. 107–132. Amsterdam University Press (2008)
4. Boneva, I., Niehren, J., Sakho, M.: Regular matching and inclusion on compressed tree patterns with context variables. In: Martín-Vide, C., Okhotin, A.,

Shapira, D. (eds.) Language and Automata Theory and Applications - 13th International Conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11417, pp. 343–355. Springer (2019). https://doi.org/10.1007/978-3-030-13435-8_25, `https://doi.org/10.1007/978-3-030-13435-8_25`

5. Cai, J., Paige, R.: Using multiset discrimination to solve language processing problems without hashing. Theor. Comput. Sci. **145**(1&2), 189–228 (1995). https://doi.org/10.1016/0304-3975(94)00183-J, `https://doi.org/10.1016/0304-3975(94)00183-J`
6. Ganardi, M., Jez, A., Lohrey, M.: Balancing straight-line programs. In: Zuckerman, D. (ed.) 60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019. pp. 1169–1183. IEEE Computer Society (2019). https://doi.org/10.1109/FOCS.2019.00073, `https://doi.org/10.1109/FOCS.2019.00073`
7. Gascón, A., Lohrey, M., Maneth, S., Reh, C.P., Sieber, K.: Grammar-based compression of unranked trees. In: Fomin, F.V., Podolskii, V.V. (eds.) Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10846, pp. 118–131. Springer (2018). https://doi.org/10.1007/978-3-319-90530-3_11, `https://doi.org/10.1007/978-3-319-90530-3_11`
8. Gasieniec, L., Kolpakov, R.M., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: 2005 Data Compression Conference (DCC 2005), 29-31 March 2005, Snowbird, UT, USA. p. 458. IEEE Computer Society (2005). https://doi.org/10.1109/DCC.2005.78, `https://doi.org/10.1109/DCC.2005.78`
9. Hucke, D., Lohrey, M., Reh, C.P.: The smallest grammar problem revisited. In: Inenaga, S., Sadakane, K., Sakai, T. (eds.) String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9954, pp. 35–49 (2016). https://doi.org/10.1007/978-3-319-46049-9_4, `https://doi.org/10.1007/978-3-319-46049-9_4`
10. Jez, A.: Faster fully compressed pattern matching by recompression. ACM Trans. Algorithms **11**(3), 20:1–20:43 (2015). https://doi.org/10.1145/2631920, `https://doi.org/10.1145/2631920`
11. Lohrey, M.: Grammar-based tree compression. In: Potapov, I. (ed.) Developments in Language Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9168, pp. 46–57. Springer (2015). https://doi.org/10.1007/978-3-319-21500-6_3, `https://doi.org/10.1007/978-3-319-21500-6_3`
12. Lohrey, M., Maneth, S., Reh, C.P.: Constant-time tree traversal and subtree equality check for grammar-compressed trees. Algorithmica **80**(7), 2082–2105 (2018). https://doi.org/10.1007/s00453-017-0331-3, `https://doi.org/10.1007/s00453-017-0331-3`
13. Maneth, S., Peternek, F.: Constant delay traversal of compressed graphs. In: Bilgin, A., Marcellin, M.W., Serra-Sagristà, J., Storer, J.A. (eds.) 2018 Data Compression Conference, DCC 2018, Snowbird, UT, USA, March 27-30, 2018. pp. 32–41. IEEE (2018). https://doi.org/10.1109/DCC.2018.00011, `https://doi.org/10.1109/DCC.2018.00011`
14. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theor. Comput. Sci. **410**(8-10), 900–913 (2009).

https://doi.org/10.1016/j.tcs.2008.12.016, `https://doi.org/10.1016/j.tcs.2008.12.016`

15. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988). https://doi.org/10.1137/0217079, `https://doi.org/10.1137/0217079`