Constant-Time Dynamic Enumeration of Word Infixes in a Regular Language

Antoine Amarilli
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Sven Dziadek
SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France

Luc Segoufin
Inria & ENS Paris, France

Abstract

For a fixed regular language L, the *enumeration of L-infixes* is the following task: we are given an input word $w = a_1 \cdots a_n$ and we must enumerate the infixes of w that belong to L, i.e., the pairs $i \leq j$ such that $a_i \cdots a_j \in L$. We are interested in *dynamic enumeration of L-infixes*, where we must additionally support letter substitution updates on w (e.g., "replace the i-th letter of w by a letter a"). Each update changes the set of infixes to enumerate, and resets the enumeration state.

We study for which regular languages L we can perform dynamic enumeration of L-infixes in constant delay (i.e., the next infix is always produced in constant time) and constant additional memory throughout the enumeration, while supporting each update in constant time.

We show that, for languages L with a neutral letter, if the language L belongs to the class ZG and is *extensible* (i.e., if $u \in L$ and u is a factor of v then $v \in L$), then dynamic enumeration of L-infixes can be achieved with a simple algorithm that ensures constant-time updates and constant delay, but not constant additional memory. Our main contribution is then to show an algorithm that additionally uses only constant additional memory, and applies to a more general class of *semi-extensible ZG* languages for which we give several equivalent characterizations. We further discuss whether our results can be generalized to larger language classes and show some (conditional) lower bounds.

2012 ACM Subject Classification Theory of computation → Regular languages

Keywords and phrases regular language, dynamic membership, enumeration, infix, ZG, automata

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.24

Related Version *Full Version*: `https://arxiv.org/abs/2602.14748` [4]

Funding *Antoine Amarilli*: Partially supported by the ANR project EQUUS ANR-19-CE48-0019 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758. *Sven Dziadek*: Supported by the ANR project STeP2 ANR-22-EXES-0013. Partially supported by the ANR project EQUUS ANR-19-CE48-0019 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758.

Acknowledgements We would like to thank the reviewers for their valuable feedback.

1 Introduction

Given a regular language L and a word $w = a_1 \cdots a_n$ the *L-infixes* of w are the pairs $[i, j]$ of w such that the factor $a_i \cdots a_j$ belongs to L. We study in this paper how to compute all L-infixes of w in the framework of *enumeration algorithms*: we distinguish between the *preprocessing time* required to read the input and compute the first result, the *delay* between any two successive results, and the *additional memory* used throughout the enumeration (i.e., the memory which is written during the enumeration phase). We always assume in this paper that the regular language L is fixed: we measure complexity only as a function of the length n of the input word, and do not study how the complexity depends on L. In

this setting, the best possible efficiency guarantees for an enumeration algorithm is to ensure that it is *linear preprocessing and constant delay* [15, 16], i.e., its preprocessing time is linear in n and that its delay is independent from n.

Enumeration algorithms have been studied in many settings [25] and in particular in database theory [22], where they have been used to efficiently enumerate query answers. In particular, as L-infixes can be expressed in MSO, it follows from [7] that there are efficient enumeration algorithms to compute L-infixes for any regular language L. Other enumeration algorithms have been designed in several special cases, in particular for *document spanners* [12, 3], a formalism for declarative information extraction [11]: these algorithms can in particular be used to enumerate L-infixes while also ensuring that the complexity remains reasonable as a function of the size of an automaton representing L.

However, one drawback of these enumeration algorithms is that their linear-time preprocessing phase must be re-run from scratch whenever the word is modified. This has motivated the investigation of enumeration algorithms on dynamic data, which aim at enumerating solutions while supporting efficient updates to the input word. In this setting, whenever the input word is modified, the enumeration of the infixes is restarted from scratch. Dynamic enumeration was first studied for MSO queries over trees [18], supporting polylogarithmic delay and update time. Finer bounds on the evaluation of MSO over words were then shown in [19]: for any fixed MSO query, given a word w of length n, one can compute in $O(n)$ a data structure allowing constant delay enumeration of the query results with constant auxiliary memory, and supporting $O(\log n)$ update time (allowed updates are relabeling, insertions, and deletions). This implies that the same bounds hold for the dynamic enumeration of L-infixes for any fixed regular language L. Dynamic enumeration was later studied in the context of document spanners, with support for SLP-represented documents and with complex editing operations [21]; and also for the specific task of pattern matching, with algorithms that achieve efficiency also in the language L describing the searched pattern [1, 6].

While linear preprocessing, constant delay, and constant auxiliary memory are hard to beat, it is not clear whether the logarithmic update time is the best possible. It turns out that for some regular languages, the answer is essentially yes: there are regular languages L admitting an unconditional $\Omega(\log n / \log \log n)$ lower bound in the cell probe model for the *dynamic membership* task of maintaining whether a word belongs to L under letter substitution updates ([23], using results from [13]; see also [5]). This can be easily transformed into a similar lower bound for the dynamic enumeration of L-infixes for some regular languages L. However, there are also regular languages L for which the dynamic enumeration problem can be solved more efficiently, i.e. with linear preprocessing time, constant delay, constant update time, and constant auxiliary memory. Our goal in this paper is to identify the regular languages for which dynamic enumeration for L-infixes enjoys this favorable complexity regime (with letter substitution updates).

Contributions. We introduce a natural property on regular languages, called being *extensible*: L is extensible if whenever $u \in L$ then $sut \in L$ for all words s and t. Extensible regular languages are an interesting class of languages to study for L-infix enumeration, because extensibility ensures that any infix that extends an L-infix is itself an L-infix. It further turns out that, for extensible languages L, the dynamic enumeration problem for L-infixes is at least as hard as the dynamic membership problem for L. Hence, focusing on regular languages with a so-called *neutral letter* (i.e., a letter having no influence on membership to the language, which is a common simplifying assumption [17, 8]), the good candidates for tractable dynamic enumeration of L-infixes are the languages from the class ZG studied in [5]: these are (conditionally) the only languages with neutral letters for which dynamic membership is feasible in constant time per update.

Our first contribution is to show that, in the presence of a neutral letter and assuming extensibility, the tractability of ZG languages L for dynamic membership extends to dynamic enumeration of L-infixes. Specifically (Proposition 4.2): for any fixed extensible language L featuring a neutral letter, we can devise an algorithm for the dynamic enumeration problem for L-infixes that achieves linear preprocessing, constant delay enumeration, and constant-time updates. The algorithm is rather simple and proceeds using an oracle for dynamic membership to L as a black-box, together with an even-odd trick [24]. However, this algorithm forgoes the constant additional memory requirement.

Our main contribution is thus to propose a dynamic enumeration algorithm that also uses only constant additional memory. We do so for extensible ZG languages, and in fact for a slight extension that we introduce in the present paper, called *semi-extensible ZG languages*. Intuitively, such languages relax the requirement of extensibility by enforcing it only on words that contain a sufficiently large number of occurrences of some non-neutral letter. So we show (Theorem 6.1): for any fixed semi-extensible ZG language L, we can devise an algorithm for dynamic enumeration of L-infixes that achieves linear preprocessing, constant delay enumeration, constant-time updates, and constant additional memory.

The key data structure involved in the proof of Theorem 6.1 is rather simple: it consists in maintaining for each letter an unordered list of all its occurrences in the word, called *occurrence lists*. This can be done using well-known data structures, see for instance the dynamic membership algorithms for ZG languages in [5]). The enumeration algorithm based on these occurrences lists is however rather technical. It performs two kinds of background traversal processes (which are amortized over the rest of the algorithm) and memorization of a carefully designed constant amount of *left information* and *right information* in relation to a pointer that sweeps over the word.

We finally show that semi-extensible ZG languages are not the only languages admitting tractable dynamic enumeration by showing other cases with ad-hoc algorithms: e.g., the non-semi-extensible ZG language $L = (aa)^*a$, or $L = b^*a$ and $L = a\Sigma^*a$ (which are non-extensible languages outside of ZG). We show however that there are (non-semi-extensible) ZG languages L that do not admit dynamic enumeration of L-infixes with constant-time updates and constant delay enumeration, subject to the *prefix-U_1* complexity hypothesis of [5]. All of this justifies that the constant-time tractability boundary for dynamic infix enumeration is in fact incomparable with the class ZG.

Paper structure. We give preliminaries and formally introduce the task of dynamic infix enumeration in Section 2. We briefly review the results of [5] in Section 3. We present our simple algorithm for extensible ZG languages (Proposition 4.2) in Section 4. We then define the class of semi-extensible ZG languages in Section 5. We give our constant additional memory algorithm for semi-extensible ZG languages (Theorem 6.1) in Section 6. We discuss other tractable cases and lower bounds in Section 7, and conclude in Section 8. Most detailed proofs are omitted due to space constraints and can be found in the full version [4].

2 Preliminaries and Problem Statement

We fix a finite alphabet Σ. We write Σ^* for the set of words over Σ, write $|w|$ for the length of a word $w \in \Sigma^*$, and write ϵ for the empty word. For $a \in \Sigma$ and $w \in \Sigma^*$, we write $|w|_a$ for the number of occurrences of a in w. We say that $u \in \Sigma^*$ is a *factor* of $v \in \Sigma^*$ if there exist $s, t \in \Sigma^*$ such that $v = sut$. A language L over an alphabet Σ is a subset of Σ^*. A letter $e \in \Sigma$ is *neutral* for a language L if, for any two words $s, t \in \Sigma^*$, we have $st \in L$ iff $set \in L$.

Infixes. Writing a word $w \in \Sigma^*$ as $w = a_1 \cdots a_n$, a *position* of w is a value i with $1 \leq i \leq n$. We write $w[i]$ to mean the i-th letter a_i of w. An *infix* of w is a 2-tuple $[i, j]$ of positions of w, with $1 \leq i \leq j \leq n$: its *left endpoint* is i, its *right endpoint* is j, and it *realizes* the word $w[i, j] \coloneqq a_i \cdots a_j$ which is a factor of w. Note that infixes always realize nonempty factors by definition, and that several infixes may realize the same factor. For $L \subseteq \Sigma^*$ a language, an *L-infix* of w is an infix realizing a word of L.

Dynamic enumeration problem. Given a language L, the problem of *dynamic enumeration for L-infixes* takes as input a word w and must build a data structure on w that will support efficient *enumeration* of all L-infixes of w and be efficiently maintainable under *letter substitution updates* on w. We will assume that L is fixed, so that complexity will only be measured as a function of the length of w (which is never changed by updates).

Formally, we consider data structures which must support two operations:

- *Letter substitution updates* on the current word w: the update is specified by a position $1 \leq i \leq |w|$ and a letter $a \in \Sigma$, and it modifies w by replacing its i-th character by a.
- *Enumeration queries*: the query asks us to output all L-infixes of the current word exactly once. We do not specify in which order the L-infixes are returned.

▶ **Example 2.1.** Consider the language $L = a^*$ and the initial word $w = aaa$. An enumeration query produces the six infixes $[1, 1]$, $[1, 2]$, $[1, 3]$, $[2, 2]$, $[2, 3]$ and $[3, 3]$ (in some order). An update operation $(2, b)$ changes w to aba. If we now perform again an enumeration query, we receive the two infixes $[1, 1]$ and $[3, 3]$. We can then perform two successive update operations $(1, b)$ and $(3, b)$, changing the word to $w = bbb$. If we now perform an enumeration query, we receive no results.

The goal of this paper is to design data structures for the dynamic enumeration problem for L-infixes which efficiently support letter substitution updates and enumeration queries. Our focus is on achieving *constant-time updates*, and *constant delay* and *constant memory enumeration*, where *constant* means independent from the length of the word w. We also require *linear preprocessing and linear memory*. Formally, we mean that there is an integer $B \in \mathbb{N}$, depending only on L, such that, for any word w:

Constant-time updates. Each update is processed in at most B computation steps.

Constant delay enumeration. While processing an enumeration query, after at most B computation steps, we produce the next L-infix (or conclude that none are left).

Constant memory enumeration. During the processing of an enumeration query, the total additional number of memory cells used is bounded by B. (The state of the memory before we started the enumeration query can still be used but in a read-only fashion.)

Linear preprocessing and linear memory. The running time to build the data structure on the initial word is bounded by $B|w|$ steps, and the total number of memory cells used by the data structure is always bounded by $B|w|$ (i.e., it never exceeds this during the lifetime of the structure, no matter how many updates or enumeration queries are performed).

We use the unit-cost RAM model [14], with cell size logarithmic in the word length $n \coloneqq |w|$ (which never changes). For instance, an integer in range $\{1, \ldots, n\}$ fits in one memory cell.

Note that the constant memory enumeration constraint implies that we can freely interrupt the enumeration process when an update arrives. Indeed, it ensures that the data structure used to support the updates is not modified during the enumeration phase, so that an incoming update can be applied at any time. However, after each update, the enumeration status is reset: enumeration queries always restart from scratch after an update, which is sensible because the set of L-infixes to enumerate may have changed.

In this paper we study the problem of dynamic enumeration of L-infixes for regular languages L. We show language families for which we can construct data structures achieving the requirements above, and languages for which this is not possible (sometimes conditionally).

3 Dynamic Membership and ZG Languages

The dynamic enumeration problem for L-infixes is related to the task of maintaining membership to L under updates, which is called *dynamic membership* and is studied in [23, 5]. In this section, we review the definition of the dynamic membership problem and some results from [23, 5], focusing on a class of regular languages called *ZG*. We recall the definition of the *syntactic monoid* of a language and define *ZG languages* as those languages whose syntactic monoid falls in a specific class of monoids, also called ZG. We then state the result from [23, 5] that ZG languages enjoy constant-time dynamic membership, and that this result is tight for languages having a neutral letter, conditionally to a complexity hypothesis.

Dynamic membership. Consider a language L. The *dynamic membership* problem is similar to the dynamic enumeration problem mentioned in the previous section, and asks us to design a data structure that can handle the same letter substitution updates. However, enumeration queries are replaced by a membership query that asks whether the current word is in L.

It was shown in [5] that, for a certain class QLZG of regular languages, there is a data structure with linear preprocessing and linear memory that can *solve dynamic membership in constant time*, i.e., both letter substitution updates and membership queries are supported in constant time. This is analogous to our goal for dynamic enumeration of L-infixes.

Further, it is shown in [5] that QLZG is the largest class of regular languages having this property, conditionally to a computational assumption called *prefix-U_1* that we will review in Section 7. The definition of the class QLZG is somewhat technical, so we will review a simpler subclass of QLZG from [5], called ZG, that is sufficient for our needs. The two classes coincide for languages with a neutral letter, which are the main focus of this paper.

ZG languages and ZG monoids. The *syntactic monoid* M of a regular language L is the monoid obtained by quotienting the words of Σ^* by the syntactic equivalence relation $\sim_L$ defined as follows: we have $u \sim_L v$ for $u, v \in \Sigma^*$ whenever we have $sut \in L$ iff $svt \in L$ for each $s, t \in \Sigma^*$. As the syntactic equivalence relation $\sim_L$ is a congruence (i.e., when $u \sim_L v$ and $u' \sim_L v'$ then $uu' \sim_L vv'$), then the concatenation of words gives a composition law on the equivalence classes of Σ^* under $\sim_L$. In particular, any neutral letter is syntactically equivalent to the empty word ϵ, which corresponds to the neutral element of M. It is well-known that the syntactic monoid of a regular language is always finite, and that any finite monoid M has an *idempotent power*, denoted ω, such that $x^\omega = x^\omega x^\omega$ for all $x \in M$.

The class of monoids *ZG* consists of all finite monoids M satisfying the equation $yx^{\omega+1} = x^{\omega+1}y$ for every $x, y \in M$. The class of languages *ZG* then consists of those languages whose syntactic monoid is in ZG. Intuitively, ZG languages are those languages where frequent letters are *central*, i.e., they commute with all other elements.

▶ **Example 3.1.** Examples of ZG languages are $L_1 = (aa)^*$, $L_2 = ab$, $L_3 = \{w \in \{a, b\}^* \mid |w|_a \geq 3 \text{ and } |w|_b \mod 3 = 0\}$, $L_4 = (a + b)^* c (a + b)^* d (a + b)^*$, and $L_5 = L_3 \cap L_4$.

Tractability for ZG languages. Let us summarize in the following proposition what is known from [5] about dynamic membership with constant time per update:

▶ **Proposition 3.2** (Follows from Theorem 1.1 in [5]). *Let L be a regular language. If L is in ZG, then the dynamic membership problem to L can be solved in constant time with linear preprocessing and linear memory. Conversely, assuming the prefix-U_1 hypothesis (see Section 7), and assuming that L features a neutral letter, if the dynamic membership problem to L can be solved in constant time, then L is in ZG.*

4 Extensible Languages and Simple Infix Enumeration Algorithm

In this section, we introduce *extensible* languages (also known as *two-sided ideals* [10]) as a simple class of languages L for which the dynamic enumeration problem of L-infixes can be reduced to the dynamic membership problem in a simple way, in the presence of a neutral letter, and up to forgoing our constant memory enumeration requirement. Let us first define the notion of extensible languages:

▶ **Definition 4.1** (Extensible language). *An* extension *of a word $w \in \Sigma^*$ is any word $z \in \Sigma^*$ such that w is a factor of z, i.e., we have $z = swt$ for some choice of $s, t \in \Sigma^*$. A language L is* extensible *if, for any words $w, z \in \Sigma^*$, if $w \in L$ and z is an extension of w then $z \in L$.*

Equivalently, a language L is extensible iff $L = \Sigma^* L \Sigma^*$. The key property of extensible languages is that, by contraposition, we have $w \notin L$ iff no infix of w realizes a word of L. Hence testing membership of w to L is equivalent to testing whether the enumeration of L-infixes on w produces results.

The extensibility of a language L can be used in two directions. First, by the above, the dynamic membership problem for L immediately reduces to the dynamic enumeration problem for L-infixes. We will use this in Section 7 to show lower bounds on the latter problem. Second, if L contains a neutral letter, we can do the converse and reduce the dynamic enumeration of L-infixes to dynamic membership to L. Namely:

▶ **Proposition 4.2.** *Let L be an extensible language featuring a neutral letter. Assume that we have a data structure Ψ for dynamic membership to L with linear preprocessing, linear memory, and constant-time updates and membership queries. Then we can build a data structure for the dynamic enumeration problem for L-infixes with linear preprocessing, linear memory, constant-time updates and constant delay enumeration.*

In particular, in view of Proposition 3.2, this implies the following:

▶ **Corollary 4.3.** *Let L be an extensible ZG language. Then the dynamic enumeration problem for L-infixes can be solved with linear preprocessing, linear memory, constant-time updates, and constant delay enumeration.*

However, we do *not* achieve constant memory enumeration. During the enumeration phase, the algorithm modifies the data structure by simulating updates. It only restores it at the end of the enumeration. Even worse, our algorithm cannot perform (real) updates until the enumeration is over. We sketch the proof of Proposition 4.2 below:

Proof sketch. The data structure, denoted Ψ, is the one used to handle updates while maintaining dynamic membership to L. We start enumeration by querying Ψ to determine if the current word w belongs to L. If not, w has no L-infixes and we are done. Otherwise we can produce the infix $[1, |w|]$. Then we decrement the right endpoint and perform substitution updates (by the neutral letter) in Ψ to remove letters one by one and produce infixes of the form $[1, r]$. We stop when the current infix is no longer in L according to Ψ. Then we add back the letters of w to Ψ and increment back r until reaching $|w|$. To ensure constant

delay, we use the standard even-odd technique [24]: we produce infixes with even r as we are decrementing r and those with odd r as we are incrementing r. We continue similarly for successive values $l \geq 1$ of the left endpoint, by increasing order; again the even-odd technique can be used to ensure that Ψ is back to its initial state at the end of the enumeration. ◀

In Section 6, we provide a more complex algorithm that fulfills our constant memory enumeration, and which further applies to a larger class of *semi-extensible ZG languages*.

5 Semi-Extensible ZG Languages

We now define our class of *semi-extensible ZG languages*, which generalize extensible ZG languages. We present three equivalent definitions: one algebraic definition involving the ZG equation, one operational definition used in our algorithms, and one self-contained definition. We then give our algorithm for dynamic enumeration of L-infixes in the next section.

Algebraic definition. We give a first definition via the ZG equation (recall that ω denotes the idempotent power of M):

▶ **Definition 5.1** (Semi-extensible ZG language, algebraic def.). *A regular language L is* semi-extensible ZG *if it is in ZG, i.e. its syntactic monoid M satisfies the ZG equation:*

$$\text{for all } x, y \in M, \text{ we have } yx^{\omega+1} = x^{\omega+1}y\,, \tag{ZG}$$

and further L is semi-extensible *in the following sense:*

for all words $x, y, z \in \Sigma^*$ and all non-neutral letters a, if $y \in L$ then $xyza^\omega \in L$.

Extensibility implies semi-extensibility, so semi-extensible ZG languages generalize extensible ZG languages which we studied in the previous section. The inclusion is strict: the language L on alphabet $\Sigma = \{a, e\}$ that has neutral letter e and consists of the words with no a's or at least two a's is semi-extensible ZG but not extensible. It is also easy to deduce from the definition that semi-extensible ZG languages are always aperiodic, i.e., their syntactic monoid satisfies the equation $x^\omega = x^{\omega+1}$. We will give an example of a semi-extensible ZG language later in this section once we have stated our three equivalent definitions of the class.

Operational definition. We give another definition of semi-extensible ZG languages for our enumeration algorithms. In the definitions below, for any subalphabet $S \subseteq \Sigma$ and word $u \in \Sigma^*$, we denote by u_{-S} the word over $\Sigma \setminus S$ obtained by removing all letters in S from u. (In particular if $S = \Sigma$ then we have $u_{-S} = \epsilon$ for any u.)

▶ **Definition 5.2** (Semi-extensible ZG language, operational def.). *Let L be a regular language. For each non-empty set S of non-neutral letters for L, let $Cond(S) \subseteq \Sigma^*$ be the (regular) language of all words u for which there is a word $v \in L$ such that v_{-S} is a factor of u_{-S}.*

Then the language L is semi-extensible ZG if there is a number $p \in \mathbb{N}$, called the threshold*, satisfying the following. We require that for each word $u \in \Sigma^*$, letting T be the set of non-neutral letters a of Σ such that $|u|_a \geq p$, we have either $T = \emptyset$ or we have that:*

$$u_{-T} \in Cond(T) \text{ iff } u \in L$$

In the definition above, notice that for any non-empty set S of non-neutral letters, we have that all letters of S are neutral for $\mathrm{Cond}(S)$, and further if $u_{-S} \notin \mathrm{Cond}(S)$ then $u \notin L$. Indeed, one can show the contrapositive by taking $v = u$ in the definition of $\mathrm{Cond}(S)$.

Self-contained definition. We last give a self-contained definition:

▶ **Definition 5.3** (Semi-extensible ZG language, self-contained def.). *A regular language is semi-extensible ZG if there exists a number $p \in \mathbb{N}$ such that the following holds for all words $u, v \in \Sigma^*$: letting T be the set of non-neutral letters a of Σ such that $|u|_a \geq p$, if $v \in L$ and T is non-empty and v_{-T} is a factor of u_{-T} then $u \in L$.*

We give an example illustrating the power of semi-extensible ZG languages:

▶ **Example 5.4.** Consider the language L_{ab} on alphabet $\Sigma = \{a, b, e\}$ that has e as a neutral letter and contains the words $u \in \Sigma^*$ satisfying one of the following: (1.) $u_{-\{e\}} = ab$ (i.e., u is exactly the word ab up to the neutral letters); or (2.) $|u|_a \geq 3$ (u has a least 3 a's); or (3.) $|u|_b \geq 3$ (u has at least 3 b's). The language L_{ab} is not extensible (e.g., $ab \in L_{ab}$ but $abb \notin L_{ab}$), but it is semi-extensible ZG with threshold $p = 3$. Indeed, by the characterization of Definition 5.3, let $v, u \in \Sigma^*$ be words such that $v \in L_{ab}$, such that $T = \{a \neq e \mid |u|_a \geq p\}$ is non-empty, and such that v_{-T} is a factor of u_{-T}. We have $u \in L_{ab}$ because u contains at least 3 occurrences of a non-neutral letter (i.e., from the nonempty set T).

The languages Cond in Definition 5.2 for L_{ab} are simply $\mathrm{Cond}(\{a, b\}) = \mathrm{Cond}(\{a\}) = \mathrm{Cond}(\{b\}) = \Sigma^*$ because all words are in L_{ab} when some non-neutral letter occurs ≥ 3 times.

Equivalence of definitions. We can show that the three definitions are equivalent:

▶ **Proposition 5.5.** *Definitions 5.1, 5.2, and 5.3 define the same family of languages.*

Properties of semi-extensible ZG languages. We close the section with two immediate properties that we will use in our algorithms in the next section. We focus on the operational definition, Definition 5.2, which we use in the sequel.

Based on Definition 5.2, let L be a semi-extensible ZG language, and pick a suitable threshold $p \in \mathbb{N}$. Up to increasing p, we assume without loss of generality that $p \geq 2$. When considering a word $w \in \Sigma^*$, we will partition the alphabet Σ into three kinds of letters: the *neutral* letters (which have no impact for membership to L), the *rare* letters $a \in \Sigma$ such that $|w|_a < p$, and the *frequent* letters $a \in \Sigma$ such that $|w|_a \geq p$.

We first observe that membership to the languages $\mathrm{Cond}(S)$ in Definition 5.2 can be verified in constant time on words with a small number of non-neutral letters:

▶ **Lemma 5.6.** *Consider a semi-extensible ZG language L with threshold p. We can precompute a data structure allowing us to do the following in constant time: given a non-empty set $S \subseteq \Sigma$ of non-neutral letters, given a word w on alphabet $\Sigma \setminus S$ with at most p occurrences of each letter, determine in constant time whether $w \in \mathrm{Cond}(S)$.*

Proof sketch. We simply tabulate explicitly the set $\mathrm{Cond}(S)$ on words with at most p occurrences of each non-neutral letter, as their size is constant. ◀

Second, we show that membership to $\mathrm{Cond}(T)$, for T the set of frequent letters of a word w, is a necessary condition for w to contain factors belonging to L. This allows us to stop enumeration early whenever the current infix realizes a factor $w \notin \mathrm{Cond}(T)$:

▶ **Lemma 5.7.** *Consider a semi-extensible ZG language L. Let w be a word and T be the set of frequent letters of w. If $T \neq \emptyset$ and $w \notin \mathrm{Cond}(T)$, then no factor of w belongs to L.*

6 Infix Enumeration Algorithm for Semi-Extensible ZG Languages

Having defined semi-extensible ZG languages, we show the following in this section. This is the main technical result of the paper, and generalizes Corollary 4.3:

▶ **Theorem 6.1.** *Let L be a semi-extensible ZG language. Then the dynamic enumeration problem of L-infixes can be solved with constant-time updates, constant delay enumeration, constant-memory enumeration, linear preprocessing, and linear memory.*

In the rest of this section, we prove Theorem 6.1. We let w be the word on which the algorithm runs, and $n := |w|$ be its length, which never changes. We first present the preprocessing and the support for update operations in Section 6.1, with the classical data structure of *occurrence lists*. The main technical challenge is to support L-infix enumeration queries with constant delay and constant memory, which we describe in Section 6.2.

6.1 Preprocessing and Updates with Occurrence Lists

Occurrence lists are a standard data structure which store elements in an array of size n, coupled with a doubly-linked list to efficiently iterate over the elements in insertion order (similar, e.g., to the LinkedHashSet data structure in Java). Formally:

▶ **Definition 6.2.** *An* occurrence list *is a data structure Λ that stores a set S of integers of $\{1, \ldots, n\}$ and supports the following operations:*

- *Insert: given $i \in \{1, \ldots, n\} \setminus S$, add i to S;*
- *Delete: given $i \in S$, remove i from S;*
- *Count: get the size $|S|$ of S;*
- *Retrieve: enumerate all the elements currently in S.*

▶ **Proposition 6.3** (Folklore)**.** *We can implement occurrence lists to take memory $O(n)$ and to ensure that all operations take $O(1)$ computation steps, except the Retrieve operation which enumerates the elements with $O(1)$ delay.*

We clarify that the data structure stores elements in insertion order and not in increasing order, so in particular it cannot be used as a constant-time priority queue.

Preprocessing and updates. We can now explain how occurrence lists are used for the algorithm that we use to show Theorem 6.1. The algorithm will use an occurrence list Λ_a for each letter $a \in \Sigma$, to store the set of positions of w that contain the letter a. We denote by Λ the tuple consisting of all occurrence lists Λ_a for $a \in \Sigma$.

The preprocessing of the algorithm simply consists of building Λ on the input word w, in linear time and with linear memory by Proposition 6.3. The handling of letter substitution updates simply amounts to maintaining Λ up to date. Formally, let (i, a) be a letter substitution update, and let $b \in \Sigma$ be the letter $w[i]$ before the update (which can be obtained in constant time and linear memory up to storing the current w in a table). We handle the update by applying it to Λ, namely, we remove i from Λ_b and add i to Λ_a, in constant time by Proposition 6.3.

6.2 Main Enumeration Algorithm

We now present how our algorithm handles L-infix enumeration queries. The overall enumeration pseudocode is given in Algorithm 1 (with some helper functions explained later).

Algorithm 1 Enumerate L-infixes of a word.

Input: semi-extensible ZG language L, threshold $p \geq 2$, languages $\mathrm{Cond}(S)$ for each set of non-neutral letters S (see Definition 5.2), word w, occurrence lists Λ_a for $a \in \Sigma$,
Output: enumeration of the L-infixes of w

```
1: nocc, nocc′ ← empty arrays
2: for all a ∈ Σ do
3:     nocc[a] ← |Λ_a|                                   /* number of occurrences from occurrence lists */
4:     StartMinBackgroundTraversal(a, Λ_a)
5: 𝔏_{-1}, ℜ_{-1} ← Unset
6: for l ← 1 to |w| do
7:     r ← |w|
8:     nocc′ ← copy of nocc
9:     ℜ ← InitRI( )                                     /* initialize current RI */
10:    /* initialize Δ for all rare letters                                          */
11:    Δ ← GetRareOccurrences(l, r, 𝔏_{-1}, ℜ_{-1}, nocc′)
12:    T ← {a ∈ Σ | a non-neutral and nocc′[a] ≥ p}
13:    while T ≠ ∅ and word stored in Δ is in Cond(T) do   /* relies on Lemma 5.6 */
14:        Output([l, r])                                /* as w[l, r] ∈ L */
15:        a ← w[r]                                      /* this occurrence of a will exit the right endpoint */
16:        nocc′[a] ← nocc′[a] − 1
17:        T ← {a ∈ Σ | a non neutral and nocc′[a] ≥ p}
18:        if nocc′[a] == p − 1 and a non-neutral then
19:            /* letter a just became rare; populate Δ for a                        */
20:            Δ[a] ← GetRareOccurrencesSingle(a, l, r − 1, 𝔏_{-1}, ℜ_{-1})
21:        if nocc′[a] < p − 1 and a non-neutral then
22:            Δ[a] ← Δ[a] \ {r}                         /* remove the occurrence of a from occurrence list Δ[a] */
23:        ℜ ← UpdateRI(ℜ, r)
24:        r ← r − 1                                     /* move right endpoint r one step left */
25:    if T ≠ ∅ then      /* case no-cond: by Lemma 5.7 no more L-infixes for left endpoint l */
26:        if r == |w| then
27:            /* while-loop exited immediately, so we must finish directly          */
28:            return                                    /* exit as there are no more L-infixes at all by Lemma 5.7 */
29:    else           /* case all-rare: there may be more infixes to produce with left endpoint l */
30:        if r == |w| then
31:            /* while-loop exited immediately, so we must finish directly          */
32:            ProduceRemaining(l, |w|, Δ)               /* enum infixes [i, j] with l ≤ i ≤ j ≤ |w| */
33:            return                                    /* all remaining infixes produced */
34:        else
35:            /* there may be more infixes [l, r′] to produce with r′ ≤ r           */
36:            ProduceRemainingL(l, r, Δ)                /* enum infixes [l, j] with l ≤ j ≤ r */
37:    if l == 1 then
38:        r_1 ← r + 1                                   /* store right limit r_1 for l = 1 */
39:        for all a ∈ Σ do
40:            StartMaxLeftBackgroundTraversal(a, Λ_a, r_1)
41:        𝔏 ← InitLI(r_1)                               /* initialize LI */
42:    else
43:        𝔏 ← UpdateLI(𝔏_{-1}, ℜ_{-1})                  /* compute LI at current r_l */
44:    ℜ_{-1} ← ℜ; 𝔏_{-1} ← 𝔏                            /* current LI/RI become previous LI/RI */
45:    nocc[w[l]] ← nocc[w[l]] − 1                       /* left endpoint l will move right and one letter exits */
```

High-level algorithm (blue). The core of the enumeration algorithm is highlighted in blue (also with a single blue line in the right margin) in Algorithm 1. We consider all possible infixes $[l, r]$ of w: by increasing order for l, from 1 to $|w|$ (outer for-loop); and, for each l, by decreasing order for r, from $|w|$ to l (inner while-loop). When doing so, we maintain the list T of all frequent letters within $w[l, r]$ and an occurrence list Δ of all rare letters within $w[l, r]$ (this is the main technical difficulty that will be discussed below). The while-loop runs until T is empty or $w[l, r]_{-T} \notin \mathrm{Cond}(T)$ (the condition on line 13, that can be tested in constant time by Lemma 5.6). Until then, by Definition 5.2, $[l, r]$ is an L-infix and we output it.

When exiting the while-loop, two cases may arise. The first case, called case *no-cond*, is when $T \neq \emptyset$ but $w[l, r]_{-T} \notin \mathrm{Cond}(T)$. At this point, by Lemma 5.7, we know that no smaller r can yield an L-infix for this value of l, and we can safely move on to $l + 1$. (In the case where $r = |w|$, by Lemma 5.7 we can stop the whole process, cf line 28.)

The second case, called case *all-rare*, is when $T = \emptyset$ (line 30), i.e., all non-neutral letters are now rare in $w[l, r]$. The key point is that in this case Δ contains all the non-neutral letters of $w[l, r]$ so enumerating the desired infixes is rather simple. For technical reason we consider two subcases. When $r == |w|$ (cf. line 31), we can use Δ to complete the whole enumeration. When $r < |w|$, (cf. line 35), we use Δ to produce all infixes of the form $[l, r']$ with $r' \leq r$ and we then move on to $l + 1$.

There are two remaining challenges for the algorithm. The first one is described in the next paragraph, and the second is sketched in the rest of this section.

Challenge 1: Maintaining occurrence counts (green). The first challenge is that we must keep track, for each considered infix $[l, r]$, of the set T of frequent letters (i.e., non-neutral letters that occur $\geq p$ times). Achieving this is completely routine, and is highlighted in green (also with a double green line in the right margin) in Algorithm 1. At the beginning, when $w[l, r] = w$, we initialize T using Λ (cf. line 3) which has been computed and maintained in Section 6.1. Then we create a copy for each new value of l (cf. line 8), and decrement the count for the letter $w[r]$ when r decreases (cf. line 16), and decrement the counts for the letter $w[l]$ when l increases (cf. line 45).

This way, we maintain the set T of the frequent letters in the current infix; this is used in particular to know to which language $\mathrm{Cond}(T)$ the condition of the while-loop applies.

Challenge 2: Keeping track of the rare letter occurrences. The second challenge, which is far harder, is that we must know the positions of the rare letters in the current infix $w[l, r]$ (and store them in occurrence lists Δ). This is used to test membership to $\mathrm{Cond}(T)$, and it is used when enumerating remaining infixes in case *all-rare*.

The lists Δ are small, because each rare letter occurs $< p$ times, so the memory usage of Δ is constant overall. However, finding the positions themselves is our main technical challenge, which is addressed by the non-highlighted code in Algorithm 1. We first explain the case of the first iteration of the for-loop ($l = 1$), and then explain the case $l > 1$.

Finding rare letters for $l = 1$. We explain how to obtain the occurrence list Δ_a of a non-neutral letter a that becomes rare in $w[l, r]$ in the first iteration of the for-loop, i.e., $l = 1$. The letter a may be *initially rare*, i.e., already when $r == |w|$ (line 11), or it may become rare when r is decremented (line 20). The case of an initially rare letter will in fact be subsumed by the case of a letter that becomes rare, so we only discuss the latter.

When a becomes rare (line 20) for some value of r, we need to find $p - 1$ occurrences of a in $w[1, r]$ in order to build Δ_a. This is challenging because the algorithm has not examined $w[1, r]$ yet. We can do it by traversing the occurrence list Λ_a of a in w (which was computed

during the preprocessing and maintained under letter substitution updates, see Section 6.1) and finding the $p-1$ smallest a's of w. However, as Λ_a is unsorted and has size $|w|_a$ (which is not constant), this would take too long. For this reason, we must do this traversal *in the background*.

More precisely, at line 4 we start a background process which traverses each occurrence list Λ_a for each $a \in \Sigma$. This is called the *min background traversal for* a: it is done by calling a function StartMinBackgroundTraversal on Λ_a, whose pseudocode we omit. From then on, we have a constant number $|\Sigma|$ of background tasks that run in parallel with the rest of the code. Namely, each min background traversal for a retrieves one element of Λ_a at each instruction of Algorithm 1, and memorizes the $\leq p-1$ smallest positions seen so far: this can be easily implemented in constant memory and constant time per step because p is constant. Then, when a becomes rare at $l = 1$ (line 20), then GetRareOccurrencesSingle will retrieve the result of the min background traversal for a.

The important point is that, whenever a becomes rare at $l = 1$, then the min background traversal for a finishes in time. This is true because, at that point, the right border r has "swept over" all occurrences of a except at most $p-1$.

▷ Claim 6.4. For $l = 1$, for the first value of r where a letter $a \in \Sigma$ becomes rare in $[1, r]$, then the min background traversal for a can be finished in constant time.

Thus, for $l = 1$ we can obtain an occurrence list Δ_a of a in $w[1, r]$ whenever a becomes rare. It is then easy to keep Δ_a up-to-date as r gets decremented and letter occurrences get removed (line 22). This ensures that we can indeed complete the iteration $l = 1$ of the for-loop: we know when to exit the while-loop, and can produce remaining infixes in lines 30–36 because there are only finitely many non-neutral letters left and we can process them in constant time. This concludes the case $l = 1$. (Note that the min background traversals are only needed for $l = 1$ and they will not be used in the sequel.)

Finding rare letters with $l > 1$. It remains to explain how to locate the occurrences of rare letters for $l > 1$, which is far more challenging. To do this, we use three ingredients: another kind of background traversal of Λ called *max-left background traversals*, the notion of *right information*, and the notion of *left information*. We now sketch these ingredients.

First, the *max-left background traversal* for each letter $a \in \Sigma$ is started at the end of the for-loop for $l = 1$ (line 40). We do it with a function StartMaxLeftBackgroundTraversal whose pseudocode we omit. Again, these $|\Sigma|$ background processes run in parallel with Algorithm 1, and they each traverse an occurrence list Λ_a over the entire word. The definition of the max-left background traversals depends on the value r_1, stored at line 38, which is the right position r just before we exited the while-loop for $l = 1$; we call this the *limit for* $l = 1$. (Note that we must have $r_1 \leq |w|$, because if the body of the while-loop is not executed at all, then we terminate immediately at lines 28 or 31, so $r_1 = |w| + 1$ is impossible.) The max-left background traversal for a aims at locating the $\leq p$ last a's which are left of the limit r_1.

Second, the *right information* (RI) at a position r of w is a constant-sized tuple describing some letter occurrences in w to the *right* of r. We memorize this information as r is decremented in Algorithm 1 in a data structure $\mathfrak{R}$, which is initialized at every for-loop iteration (line 9) and updated throughout the while-loop iteration (line 23): The intuition is that, for the last right endpoint position $r_l \leq |w|$ before we exited the while-loop for the l-th iteration of the for-loop (we call r_l the *limit for* l, generalizing r_1 from the previous paragraph), then the RI at r_l contains some *tracked* positions of w to the right of r_l. In particular it will include some positions that will be to the left of the position r_{l+1} where we will exit the while-loop for the next iteration of the for-loop.

▶ **Definition 6.5.** *Let w be a word and r be a position of w. The* right information *(RI) of w at r consists of the following for each letter $a \in \Sigma$:*

- *The sequence $r \leq \mu_{a,1} < \cdots < \mu_{a,n_a}$ of the $\leq p$ first a's right of r (with $n_a \leq p$): we say that the positions $\mu_{a,1}, \ldots, \mu_{a,n_a}$ are* tracked.
- *For each $1 \leq i \leq n_a$ and for each letter $b \in \Sigma$, the positions of the $\leq p$ last b's left of the tracked position $\mu_{a,i}$ and right of r.*

The RI stores the first p letters to the right of the current limit (and their predecessors) because the limit can *jump* p positions to the right for one increment of the left border l. The next example illustrates this.

▶ **Example 6.6.** Consider the language L_5 on alphabet $\Sigma = \{a, b, c, e\}$ that has e as a neutral letter and contains the words satisfying one of the following:

1. it contains at least 5 a's, at least 1 b and at least 1 c, and contains a factor $b\Sigma^*c$,
2. it contains at least 1 a, at least 5 b's, and at least 1 c, and contains a factor $c\Sigma^*a$,
3. it contains at least 1 a, at least 1 b, and at least 5 c's, and contains a factor $a\Sigma^*b$,
4. two of the letters $\{a, b, c\}$ occur at least 5 times and the remaining at least once.

The language L_5 is ZG (intuitively because the order on letters only matters for rare letters) and it is extensible because each condition in the bullet points above is extensible. Hence, L_5 is in particular semi-extensible ZG, and we can take threshold $p = 5$.

Consider now the following input word w with neutral letters already omitted.

index :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$w =$	a	b	b	b	a	a	a	a	c	c	c	c	b	c	a	a

Note that for words where the letter a is frequent ($|w|_a \geq 5$), the language $\mathrm{Cond}(\{a\})$ contains all words that contain at least one b and one c and where the b occurs before the c. This condition $\mathrm{Cond}(\{a\})$ is valid for $w[1, 9]$, but no longer valid for $w[1, 8]$ as there are no c. Thus, the first limit will be at position $r_1 = 9$.

For the left border $l = 2$, when $r \leq 14$, the word now only contains 4 a's. The smallest factor $w[2, r]$ that contains at least a frequent letter is $w[2, 14]$. There, the letter c is frequent and $\mathrm{Cond}(\{c\})$ contains all words with at least one a and one b and where the a occurs before the b. Thus, $\mathrm{Cond}(\{c\})$ is fulfilled and the second limit is $r_2 = 14$. Note that position r_2 is the fifth c of the ≤ 5 first c's right of r_1 (and it is not $|w| + 1$). So this illustrates why the RI has to store so many positions.

We now explain the third and final ingredient. For $r \geq r_1$ with r_1 the limit for $l = 1$, the *left information* (LI) at a position r of w is a constant-sized tuple of letter occurrences in w to the *left* of r, but to the right of r_1. (Intuitively, the positions to the left of r_1 are handled by the max-left background traversals.)

▶ **Definition 6.7.** *Let r_1 be the position where the while-loop exits for $l = 1$. The* left information *(LI) of w at the limit r_l (for $l \geq 1$), contains, for each letter $a \in \Sigma$, the $\leq p$ last a's left of r_l and right of r_1.*

The LI will only be used for values of r that are limits, i.e., where we exit the while-loop. For $l = 1$, we initialize the LI at r_1 at the end of the outer for-loop ($l = 1$) at line 41: this is easy because the LI for $l = 1$ concerns positions of w to the left of r_1 and to the right of r_1 (i.e., only in r_1). Then, we update the LI at r_l at the end of the l-th for-loop iteration (at line 43) from the LI at r_{l-1} and the RI at r_{l-1}.

All that remains is to show how rare letter occurrences can be properly computed whenever a letter becomes rare for $l > 1$. This is the object of the claim below (where we have to omit the proof due to space constraints):

▷ Claim 6.8. By maintaining the RI at the successive values of r, the RI at the limit r_{l-1} where we exited the for-loop at $l-1$, and the LI at r_{l-1}, together with the results of the max-left background traversals (when available), we can determine in constant time, for each $l > 1$ and letter $a \in \Sigma$, an occurrence list Δ_a of the occurrences of a in $w[l, r]$ whenever a becomes rare (i.e., either initially at line 11, or in the while-loop at line 20). This can be done while obeying the constant delay and constant enumeration memory requirements.

This concludes the presentation of Algorithm 1, and finishes the proof of Theorem 6.1.

7 Discussion and Lower Bounds

We have shown that semi-extensible ZG languages L enjoy dynamic enumeration of L-infixes with linear preprocessing and memory, constant-time updates, constant delay, and constant enumeration memory. Languages with such an algorithm are called *tractable*. In this section, we study which other languages are tractable and which are (conditionally) not tractable.

For simplicity, in the rest of this section, we focus on languages featuring a neutral letter, which we write e and omit from regular expressions (e.g., b^*a should be understood as $(e+b)^*ae^*$). We first give prerequisites about the *prefix-U_1 problem* used as a hypothesis in almost all of our lower bounds. We then discuss the case of languages outside of ZG, and then discuss the case of languages which are in ZG but not semi-extensible ZG.

Prefix-U_1. Most lower bounds presented here are conditional to the *prefix-U_1 hypothesis* from [5], that we now recall. Let $n \in \mathbb{N}$. The *prefix-U_1 problem* asks us for a data structure such that, given a set $S \subseteq \{1, \ldots, n\}$, we can perform the following tasks on input $i \leq n$: (1.) test whether $i \in S$; (2.) insert i to S; (3.) remove i from S; (4.) return YES if $i \leq \min S$.

The *prefix-U_1 hypothesis* states that there is no data structure that can achieve all these tasks in constant time (independent from n) in the unit cost RAM model with logarithmic cell size. Note that the first three operations are easy to support in constant time (e.g., with a table of Booleans), so the challenge is to also support (4.). The prefix-U_1 problem is a weaker version of the well-known *priority queue* problem (where we would replace (4.) by "return $\min S$"). The priority queue problem is itself a special case of the *successor search* problem where (4.) would be: "return $\min\{j \in S, i \leq j\}$". There are data structures solving successor search in time $O(\log \log n)$ and this has been shown to be optimal (cf. [20] and [5, Appendix A.2]). The upper bound thus also covers the priority queue problem and prefix-U_1 problem as special cases; however, no lower bounds are known for these problems specifically.

Languages outside of ZG. It turns out that there are non-ZG languages L for which dynamic enumeration of L-infixes is tractable (which is not covered by our Theorem 6.1), even though dynamic membership to L is not tractable by Proposition 3.2. For instance:

▷ Claim 7.1. The language $L_0 \coloneqq b^*a$ is not in ZG but is tractable.

Proof sketch. We maintain an occurrence list Λ_a of the a's in the word under updates, and enumerate L_0-infixes by considering every a and considering all ways to move the endpoints (over neutral letters, or over b's for the left endpoint). ◁

By contrast, the following similar language $L_1 \coloneqq b^+a$ is (conditionally) intractable:

▷ Claim 7.2. Under the prefix-U_1 hypothesis, the language $L_1 \coloneqq b^+a$ is not tractable

Proof sketch. We reduce from prefix-U_1: we store positions of a set S as letter b's, and insert a letter a at a position i to test if the leftmost b is $> i$ or not. ◁

To illustrate further how sensitive the problem is, consider the following:

▷ Claim 7.3. $L_2 \coloneqq a\Sigma^*a$ is tractable, but $L_2' \coloneqq b\Sigma^*a$ is not (under the prefix-U_1 hypothesis).

Proof sketch. For L_2 we consider all pairs of a's from the occurrence lists and consider all ways to move the endpoints left and right of the chosen a's. For L_2' we do like in Claim 7.2. ◁

We close the paragraph with two remarks. First, any *extensible* regular language outside of ZG is not tractable under the prefix-U_1 hypothesis. Indeed, we explained in Section 4 that, for extensible languages, dynamic membership for L reduces to dynamic enumeration of L-infixes: thus Proposition 3.2 concludes. Second, some regular languages outside of ZG are *unconditionally* not tractable, by reducing from prefix-parity (see [13]).

Languages in ZG. We now study the case of languages L in ZG and discuss whether semi-extensibility is necessary for the language to be tractable. We first claim that some non-semi-extensible ZG languages are indeed (conditionally) not tractable:

▷ Claim 7.4. Consider the language L_3 on alphabet $\Sigma = \{a, b, c, e\}$ (with e neutral) that contains the words w such that $|w|_c \geq 1$ and $w_{-\{c\}} \in L_{ab}$, for L_{ab} the language of Example 5.4. Then L_3 is non-semi-extensible ZG and is not tractable under the prefix-U_1 hypothesis.

However, some non-semi-extensible ZG languages are tractable, for instance:

▷ Claim 7.5. The language $L_4 \coloneqq (aa)^*a$ of words with an odd number of a's is tractable.

Proof sketch. We go over the occurrence list for a and enlarge L_4-infixes around each a. ◁

Thus, a characterization of the languages with tractable infix enumeration under updates is still out of reach, even for ZG languages, indeed even for finite languages with a neutral letter. For instance we do not know the status of $(aa)^+$, or even of aa.

8 Conclusion and Future Work

We have studied the dynamic enumeration of L-infixes for fixed languages L, aiming at linear preprocessing, linear memory, constant-time updates, constant delay, and constant-memory enumeration. Our main achievement is an algorithm that applies to any fixed *semi-extensible ZG language*. Further, we have discussed other tractable and intractable cases.

One obvious direction for further research is to achieve a complete characterization of the tractable regular languages. However, the algorithm of Theorem 6.1 shows that some tractable cases are technical, while other simple cases (e.g., $L = aa$ with neutral letter) remain open. Some other tractable cases would deserve further study, e.g., languages like $a\Sigma^*b + b\Sigma^*a$ (like Claim 7.3), or languages generalizing the algorithm for $(aa)^*a$ (cf Claim 7.5).

Another research direction would be to characterize languages with different complexity regimes below $O(\log n)$. In particular, one could aim at generalizing the $O(\log \log n)$ class for dynamic membership from [5]. The question can be generalized to more general queries than L-infix queries, on words or even on trees [2]; or to more general update operations like letter insertions or deletions. Another relevant type of queries, whose complexity might

differ, is *direct access* queries [9], where we maintain under updates a data structure that allows us to directly access the i-th L-infix of the word given the index i; or *tuple testing* queries, where we can determine efficiently whether a given infix $[i, j]$ realizes a word of L. More generally, a natural question for future work is whether our results can generalize to more general queries in monadic second-order logic (beyond L-infix queries). It would also be interesting to study the alternative semantics where after each update we must enumerate only the new results, though we expect that this would require different techniques.

References

1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338645.

2 Antoine Amarilli, Corentin Barloy, Louis Jachiet, and Charles Paperman. Dynamic membership for regular tree languages. In *MFCS*, 2025. doi:10.4230/LIPIcs.MFCS.2025.8.

3 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019. doi:10.4230/LIPIcs.ICDT.2019.22.

4 Antoine Amarilli, Sven Dziadek, and Luc Segoufin. Constant-time dynamic enumeration of word infixes in a regular language, 2026. Full version with proofs. doi:10.48550/arXiv.2602.14748.

5 Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. In *ICALP*, 2021. doi:10.4230/LIPIcs.ICALP.2021.116.

6 Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *TALG*, 3(2), 2007. doi:10.1145/1240233.1240242.

7 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006. doi:10.1007/11874683_11.

8 David A Mix Barrington, Neil Immerman, Clemens Lautemann, Nicole Schweikardt, and Denis Thérien. First-order expressibility of languages with neutral letters or: The Crane Beach conjecture. *JCSS*, 70(2), 2005. doi:10.1016/j.jcss.2004.07.004.

9 Pierre Bourhis, Florent Capelli, Stefan Mengel, and Cristian Riveros. Dynamic direct access of MSO query evaluation over strings. In *ICDT*, 2025. doi:10.4230/LIPIcs.ICDT.2025.26.

10 Janusz A. Brzozowski and Yuli Ye. Syntactic complexity of ideal and closed languages. In *DLT 2011*, pages 117–128, 2011. doi:10.1007/978-3-642-22321-1_11.

11 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *JACM*, 62(2), 2015. doi:10.1145/2699442.

12 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, 2018. doi:10.1145/3196959.3196987.

13 Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *STOC*, 1989. doi:10.1145/73007.73040.

14 Étienne Grandjean and Louis Jachiet. Which arithmetic operations can be performed in constant time in the RAM model with addition?, 2022. arXiv:arXiv:2206.13851.

15 Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *LMCS*, 7, 2011. doi:10.2168/LMCS-7(2:20)2011.

16 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013. doi:10.1145/2528928.

17 Michal Kouckỳ, Pavel Pudlák, and Denis Thérien. Bounded-depth circuits: Separating wires from gates. In *STOC*, 2005. doi:10.1145/1060590.1060629.

18 Katja Losemann and Wim Martens. MSO queries on trees: Enumerating answers under updates. In *CSL–LICS*, 2014. doi:10.1145/2603088.2603137.

19 Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, 2018. doi:10.1145/3196959.3196961.

20 Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *SODA*, 2007. URL: https://dl.acm.org/doi/10.5555/1283383.1283443.

21 Markus L Schmid and Nicole Schweikardt. Query evaluation over SLP-represented document databases with complex document editing. In *PODS*, 2022. doi:10.1145/3517804.352415.

22 Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In *STACS*, 2014. doi:10.4230/LIPIcs.STACS.2014.13.

23 Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *JACM*, 44(2), 1997. doi:10.1145/256303.256309.

24 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. *National Institute of Informatics (in Japan) Technical Report E*, 4, 2003. doi:10.20736/0000000385.

25 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, 2016. arXiv:1605.05102.