Factorization Forests

Mikołaj Bojańczyk

Warsaw University

Abstract. A survey of applications of factorization forests.

Fix a regular language $L \subseteq A^*$. You are given a word $a_1 \cdots a_n \in A^*$. You are allowed to build a data structure in time $O(n)$. Then, you should be able to quickly answer queries of the form: given $i \leq j \in \{1, \ldots, n\}$, does the infix $a_i \cdots a_j$ belong to L?

What should the data structure be? What does quickly mean? There is natural solution that uses a divide and conquer approach. Suppose that the language L is recognized by a (nondeterministic) automaton with states Q. We can divide the word in two halves, then into quarters and so on. The result is a binary tree decomposition, where each tree node corresponds to an infix, and its children divide the infix into two halves. In a bottom-up pass we decorate each node of the tree with the set $R \subseteq Q^2$ of pairs (source, target) for runs over node's corresponding infix. This data structure can be computed in time linear in the length of the word. Since the height of this tree is logarithmic, a logarithmic number of steps is sufficient to compute the set R of any infix (and the value of R determines membership in L).

The goal of this paper is to popularize a remarkable combinatorial result of Imre Simon [15]. One of its applications is that the data structure above can be modified so that the queries are answered not in logarithmic time, but constant time (the constant is the size of a semigroup recognizing the language).

So, what is the Simon theorem? Let $\alpha : A^* \to S$ be a morphism into a finite monoid[1]. Recall the tree decomposition mentioned in the logarithmic divide and conquer algorithm. This tree decomposes the word using a single rule, which we call the *binary rule*: each word $w \in A^*$ can be split into two factors $w = w_1 \cdot w_2$, with $w_1, w_2 \in A^*$. Since the rule is binary, we need trees of at least logarithmic height (it is a good strategy to choose w_1 and w_2 of approximately same length). To go down to constant height, we need a rule that splits a word into an unbounded number of factors. This is the *idempotent rule*: a word w can be factorized as $w = w_1 \cdot w_2 \cdots w_k$, as long as the images of the factors $w_1, \ldots, w_k \in A^*$ are all equal, and furthermore idempotent:

$$\alpha(w_1) = \cdots = \alpha(w_k) = e \qquad \text{for some } e \in S \text{ with } ee = e.$$

[1] Recall that a monoid is a set with an associative multiplication operation, and an identity element. A morphism is a function between monoids that preserves the operation and identity.

An *α-factorization forest* for a word $w \in A^*$ is an unranked tree, where each leaf is labelled by a single letter or the empty word, each non-leaf node corresponds to either a binary or idempotent rule, and the rule in the root gives w.

Theorem 1 (Factorization Forest Theorem of Simon [15]). *For every morphism $\alpha : A^* \to S$ there is a bound $K \in \mathbb{N}$ such that all words $w \in A^*$ have an α-factorization forest of height at most K.*

Here is a short way of stating Theorem 1. Let X_i be the set of words that have an α-factorization forest of height i. These sets can be written as

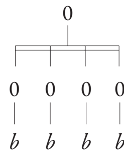
$$X_1 = A \cup \{\epsilon\} \qquad X_{n+1} = X_n \cdot X_n \cup \bigcup_{\substack{e \in S \\ ee=e}} (X_n \cap \alpha^{-1}(e))^* \ .$$

The theorem says that the chain $X_1 \subseteq X_2 \subseteq \cdots$ stabilizes at some finite level.

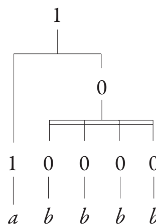
Let us illustrate the theorem on an example. Consider the morphism $\alpha : \{a, b\}^* \to \{0, 1\}$ that assigns 0 to words without an a and 1 to words with an a. We will use the name *type of w* for the image $\alpha(w)$. We will show how that any word has an α-factorization forest of height 5.

Consider first the single letter words a and b. These have α-factorization forests of height one (the node is decorated with the value under α):

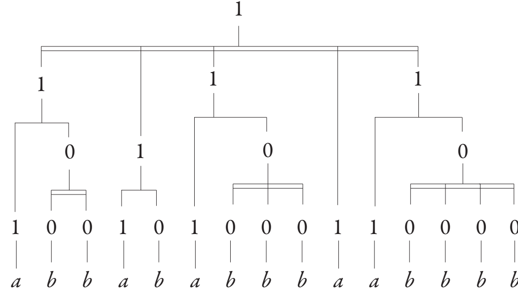
Next, consider words in b^+. These have α-factorization forests of height 2: one level is for the single letters, and the second level applies the idempotent rule, which is legal, since the type 0 of b is idempotent:



In the picture above, we used a double line to indicate the idempotent rule. The binary rule is indicated by a single line, as in the following example:



As the picture above indicates, any word in ab^+ has an α-factorization forest of height 3. Since the type of ab^+ is the idempotent 1, we can apply the idempotent rule to get a height 4 α-factorization forest for any word in $(ab^+)^+$:



This way, we have covered all words in $\{a,b\}^*$, except for words in $b^+(ab^+)^+$. For these, first use the height 4 factorization forest for the part $(ab^+)^+$, and then attach the prefix b^+ using the binary rule.

A relaxed idempotent rule. Recall that the idempotent rule requires the word w to be split into parts $w = w_1 \cdots w_k$ with the same idempotent type. What if we relaxed this rule, by only requiring all the parts to have the same type, but not necessarily an idempotent type? We claim that relaxing the idempotent rule would not make the Factorization Forest Theorem any simpler. The reason is that in any finite monoid S, there is some power $m \in \mathbb{N}$ such s^m is idempotent for any $s \in S$. Therefore, any application of the relaxed rule can be converted into a height $\log m$ tree with one idempotent rule, and a number of binary rules.

1 Proof of the theorem

This section contains a proof of the Factorization Forest Theorem, based on a proof by Manfred Kufleitner [9], with modifications suggested by Szymon Toruńczyk. The proof is self-contained. Implicitly it uses Green's relations, but these are not explicitly named.

We define the *Simon height* $\|S\|$ of a finite monoid S to be the smallest number K such that for every morphism $\alpha : A^* \to S$, all words in A^* have an α-factorization forest of height at most K. Our goal is to show that $\|S\|$ is finite for a finite monoid S. The proof is by induction on the number of elements in S. The induction base, when S has one element, is obvious, so the rest of the proof is devoted to the induction step.

Each element $s \in S$ generates three ideals: the left ideal Ss, the right ideal sS and the two-sided ideal SsS. All of these are submonoids and contain s. Elements of S are called *$\mathcal{H}$-equivalent* if they have the same left and right ideals. First, we show a lemma, which bounds the height $\|S\|$ based on a morphism $\beta : S \to T$. We use this lemma to reduce the problem to monoids where there is at most one nonzero two-sided ideal (nonzero ideals are defined later). Then we use the

lemma to further reduce the problem to monoids where $\mathcal{H}$-equivalence is trivial, either because all elements are equivalent, or because all distinct elements are nonequivalent. Finally, we consider the latter two cases separately.

Lemma 1. *Let S, T be finite monoids and let $\beta : S \to T$ be a morphism.*

$$\|S\| \leq \|T\| \cdot \max_{\substack{e \in T \\ ee=e}} \|\beta^{-1}(e)\|$$

Proof
Let $\alpha : A^* \to S$ be morphism, and $w \in A^*$ a word. We want to find an α-factorization forest of height bounded by the expression in the lemma. We first find a $(\beta \circ \alpha)$-factorization forest f for w, of height bounded by $\|T\|$. Why is f not an α-factorization? The reason is that f might use the idempotent rule to split a word u into factors $u_1, \ldots, u_n$. The factors have the same (idempotent) image under $\beta \circ \alpha$, say $e \in T$, but they might have different images under α. However, all the images under α belong to the submonoid $\beta^{-1}(e)$. Treating the words $u_1, \ldots, u_n$ as single letters, we can find an α-factorization for $u_1 \cdots u_n$ that has height $\|\beta^{-1}(e)\|$. We use this factorization instead of the idempotent rule $u = u_1 \cdots u_n$. Summing up, we replace each idempotent rule in the factorization forest f by a new factorization forest of height $\|\beta^{-1}(e)\|$. ■

For an element $s \in S$, consider the two-sided ideal SsS. The equivalence relation $\sim_s$, which collapses all elements from SsS into a single element, is a monoid congruence. Therefore, mapping an element $t \in S$ to its equivalence class under $\sim_s$ is a monoid morphism β, and we can apply Lemma 1 to get

$$\|S\| \leq \|S_{/\sim_s}\| \cdot \|SsS\| \ .$$

When can we use the induction assumption? In other words, when does this inequality above use smaller monoids on the right side? This happens when SsS has at least two elements, but is not all of S. Therefore, it remains to consider the case when for each s, the two-sided ideal SsS is either S or has either one element s. This case is treated below.

At most one nonzero two-sided ideal. From now on, we assume that all two-sided ideals are either S or contain a single element. Note that if $SsS = \{s\}$ then s is a zero, i.e. satisfies $st = ts = s$ for all $t \in S$. There is at most one zero, which we denote by 0. Therefore a two-sided ideal is either S or $\{0\}$.

Note that multiplying on the right either decreases or preserves the right ideal, i.e. $stS \subseteq sS$. We first show that the right ideal cannot be decreased without decreasing the two-sided ideal.

$$\text{if } SsS = SstS \qquad \text{then} \qquad sS = stS \tag{1}$$

Indeed, if the two-sided ideals of s and st are equal, then there are $x, y \in S$ with $s = xsty$. By applying this n times, we get $s = x^n s(ty)^n$. If n is chosen so that $(ty)^n$ is idempotent, which is always possible in a finite monoid, we get

$$s = x^n s(ty)^n = x^n s(ty)^n (ty)^n = s(ty)^n,$$

which gives $sS \subseteq stS$, and therefore $sS = stS$.

We now use (1) to show that $\mathcal{H}$-equivalence is a congruence. In other words, we want to show that if s, u are in $\mathcal{H}$-equivalent, then for any $t \in S$, the elements st, ut are $\mathcal{H}$-equivalent and the elements ts, tu are $\mathcal{H}$-equivalent. By symmetry, we only need to show that st, ut are $\mathcal{H}$-equivalent. The left ideals Sst, Sut are equal by assumption on $Ss = Su$, so it remains to prove equality of the right ideals stS, utS. The two-sided ideal $SstS = SutS$ can be either $\{0\}$ or S. In the first case, $st = ut = 0$. In the second case, $SsS = SstS$, and therefore $sS = stS$ by (1). By the same reasoning, we get $uS = utS$, and therefore $utS = stS$.

Since $\mathcal{H}$-equivalence is a congruence, mapping an element to its $\mathcal{H}$-class (i.e. its $\mathcal{H}$-equivalence class) is a morphism β. The target of β is the quotient of S under $\mathcal{H}$-equivalence, and the inverse images $\beta^{-1}(e)$ are $\mathcal{H}$-classes. By Lemma 1,

$$\|S\| \leq \|S_{/\mathcal{H}}\| \cdot \max_{\substack{s \in S \\ \beta(ss)=\beta(s)}} \|[s]_{\mathcal{H}}\|.$$

We can use the induction assumption on smaller monoids, unless: a) there is one $\mathcal{H}$-class; or b) all $\mathcal{H}$-classes have one element. These two cases are treated below.

All $\mathcal{H}$-classes have one element. Take a morphism $\alpha : A^* \to S$. For $w \in A^*$, we will find an α-factorization forest of size bounded by S. We use the name *type of* w for the image $\alpha(w)$. Consider a word $w \in A^*$. Let v be the longest prefix of w with a type other than 0 and let va be the next prefix of w after v (it may be the case that $v = w$, for instance when there is no zero, so va might not be defined). We cut off the prefix va and repeat the process. This way, we decompose the word w as

$$w = v_1 a_1 v_2 a_2 \cdots v_n a_n v_{n+1} \qquad \begin{array}{l} v_1, \ldots, v_{n+1} \in A^*, a_1 \ldots, a_n \in A \\ \alpha(v_1), \ldots \alpha(v_{n+1}) \neq 0 \quad \alpha(v_1 a_1), \ldots, \alpha(v_n a_n) = 0. \end{array}$$

The factorization forests for $v_1, \ldots, v_{n+1}$ can be combined, increasing the height by three, to a factorization forest for w. (The binary rule is used to append a_i to v_i, the idempotent rule is used to combine the words $v_1 a_1, \ldots, v_n a_n$, and then the binary rule is used to append v_{n+1}.) How do we find a factorization forest for a word v_i? We produce a factorization forest for each v_i by induction on how many distinct infixes $ab \in A^2$ appear in v_i (possibly $a = b$). Since we do not want the size of the alphabet to play a role, we treat ab and cd the same way if the left ideals (of the types of) of a and c are the same, and the right ideals of b and d are the same. What is the type of an infix of v_i? Since we have ruled out 0, then we can use (1) to show that the right ideal of the first letter determines the right ideal of the word, and the left ideal of the last letter determines the left ideal of the word. Since all $\mathcal{H}$-classes have one element, the left and right ideals determine the type. Therefore, the type of an infix of v_i is determined by its first and last letters (actually, their right and left ideals, respectively). Consider all appearances of a two-letter word ab inside v_i:

$$v_i = u_0 ab u_1 ab \cdots ab u_{m+1}$$

By induction, we have factorization forests for $u_0, \dots, u_{m+1}$. These can be combined, increasing the height by at most three, to a single forest for v_i, because the types of the infixes $bu_1a, \dots, bu_ma$ are idempotent (unless $m = 1$, in which case the idempotent rule is not needed).

There is one $\mathcal{H}$-class.[2] Take a morphism $\alpha : A^* \to S$. For a word $w \in A^*$ we define $P_w \subseteq S$ to be the types of its non-trivial prefixes, i.e. prefixes that are neither the empty word or w. We will show that a word w has an α-factorization forest of height linear in the size of P_w. The induction base, $P_w = \emptyset$, is simple: the word w has at most one letter. For the induction step, let s be some type in P_w, and choose a decomposition $w = w_0 \cdots w_{n+1}$ such that the only prefixes of w with type s are $w_0, w_0w_1, \dots, w_0 \cdots w_n$. In particular,

$$P_{w_0},\ s \cdot P_{w_1},\ s \cdot P_{w_2},\ \dots\ , s \cdot P_{w_n} \quad \subseteq \quad P_w \setminus \{s\}\ .$$

Since there is one $\mathcal{H}$-class, we have $sS = S$. By finiteness of S, the mapping $t \mapsto st$ is a permutation, and therefore the sets sP_{w_i} have fewer elements than P_w. Using the induction assumption, we get factorizations for the words $w_0, \dots, w_{n+1}$. How do we combine these factorizations to get a factorization for w? If $n = 0$, we use the binary rule. Otherwise, we observe the types of $w_1, \dots, w_n$ are all equal, since they satisfy $s \cdot \alpha(w_i) = s$, and $t \mapsto st$ is a permutation. For the same reason, they are all idempotent, since

$$s \cdot \alpha(w_1) \cdot \alpha(w_1) = s \cdot \alpha(w_1) = s.$$

Therefore, the words $w_1, \dots, w_n$ can be joined in one step using the idempotent rule, and then the words w_0 and w_{n+1} can be added using the binary rule.

Comments on the proof. Actually $\|S\| = 3|S|$. To get this bound, we need a slightly more detailed analysis of what happens when Lemma 1 is applied (omitted here). Another important observation is that the proof yields an algorithm, which computes the factorization in linear time in the size of the word

2 Fast string algorithms

In this section, we show how factorization forests can be used to obtain fast algorithms for query evaluation. The idea[3] is to use the constant height of factorization forests to get constant time algorithms.

2.1 Infix pattern matching

Let $L \subseteq A^*$ be a regular language. An *L-infix query* in a word w is a query of the form "given positions $i \leq j$ in w, does the infix $w[i..j]$ belong to L?'

Below we state formally the theorem which was described in the introduction.

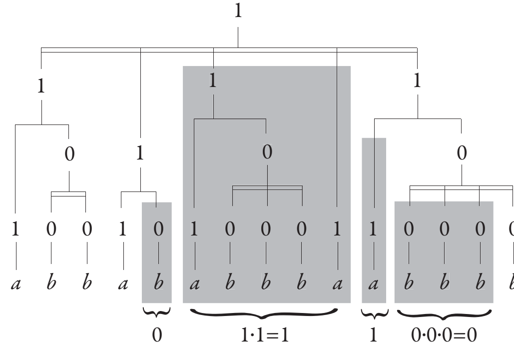
[2] Actually, in this case the monoid is a group.
[3] Suggested by Thomas Colcombet.

Theorem 2. *Let $L \subseteq A^*$ be a language recognized by $\alpha : A^* \to S$. Using an α-factorization forest f for a word $w \in A^*$, any L-infix query can be answered in time proportional to the height of f.*

Note that since f can be computed in linear time, the above result shows that, after a linear precomputation, infix queries can be evaluated in constant time. The constants in both the precomputation and evaluation are linear in S.

Proof

The proof is best explained by the following picture, which shows how the type of any infix can be computed from a constant number of labels in the factorization forest:



Below follows a more formal proof. We assume that each position in the word contains a pointer to the leaf of f that contains letter in that position. We also assume that each node in f comes with the number of its left siblings, the type of the word below that node, and a pointer to its parent node.

In the following x, y, z are nodes of f. The distance of x from the root is written $|x|$. We say a node y is *to the right* of a node x if y is not a descendant of x, and y comes after x in left-to-right depth-first traversal. A node y is *between x and z* if y is to the right of x and z is to the right of y. The word $bet(x, y) \in A^*$ is obtained by reading, left to right, the letters in the leaves between x and y. We claim that at most $|x|+|y|$ steps are needed to calculate the type of $bet(x, y)$. The claim gives the statement of the theorem, since membership in L only depends on the type of a word. The proof of the claim is by induction on $|x| + |y|$.

Consider first the case when x and y are siblings. Let $z_1, \ldots, z_n$ be the siblings between x and y. We use $sub(z)$ for the word obtained by reading, left to right, the leaves below z. We have

$$bet(x, y) = sub(z_1) \cdots sub(z_n) \ .$$

If $n = 0$, the type of $bet(x, y)$ is the identity in S. Otherwise, the parent node must be an idempotent node, for some idempotent $e \in S$. In this case, each $sub(z_i)$ has type e and by idempotency the type of $bet(x, y)$ is also e.

Consider now the case when x and y are not siblings. Either the parent of x is to the left of y or x is to the left of the parent of y. By symmetry we consider

only the first case. Let z be the parent of x and let $z_1, \ldots, z_n$ be all the siblings to the right of x. We have

$$bet(x, y) = sub(z_1) \cdots sub(z_n) \cdot bet(z, y)$$

As in the first case, we can compute the type of $sub(z_1) \cdots sub(z_n)$ in a single step. The type of $bet(z, y)$ is obtained by induction assumption. ∎

The theorem above can be generalized to more general queries than infix queries[4]. An *n-ary query* Q for words over an alphabet A is a function that maps each word $w \in A^*$ to a set of tuples of word positions $(x_1, \ldots, x_n) \in \{1, \ldots, |w|\}^n$. We say such a query Q can be *evaluated with linear precomputation and constant delay* if there is an algorithm, which given an input word w:

- Begins by doing a precomputation in time linear in the length of w.
- After the precomputation, starts outputting all the tuples in $Q(w)$, with a constant number of operations between tuples.

The tuples will be enumerated in lexicographic order (i.e. first sorted left-to-right by the first position, then by the second position, and so on).

One way of describing an n-ary query is by using a logic, such as monadic second-order logic. A typical query would be: "the labels in positions $x_1, \ldots, x_n$ are all different, and for each $i, j \in \{1, \ldots, n\}$, the distance between x_i and x_j is even". By applying the ideas from Theorem 2, one can show:

Theorem 3. *An query definable in monadic second-order logic can be evaluated with linear precomputation and constant delay.*

2.2 Avoiding factorization forests

Recall that the constants in Theorem 2 were linear in the size of the monoid S. If, for instance, the monoid S is obtained from an automaton, then this can be a problem, since the translation from automata (even deterministic) to monoids incurs an exponential blowup. In this section, we show how to evaluate infix queries without using monoids and factorization forests.

Theorem 4. *Let $L \subseteq A^*$ be a language recognized by a deterministic automaton with states Q. For any word $w \in A^*$, one can calculate a data structure in time $O(|Q| \cdot |w|)$ such that any L-infix query can be answered in time $O(|Q|)$.*

It is important that the automaton is deterministic. There does not seem to be any easy way to modify the construction below to work for nondeterministic automata.

Let the input word be $w = a_1 \cdots a_n$. A *configuration* is a pair $(q, i) \in Q \times \{0, \ldots, n\}$, where i is called the *position* of the configuration. The idea is that (q, i) says that the automaton is in state q between the letters a_i and a_{i+1}. The *successor* of a configuration (q, i), for $i < n$, is the unique configuration on

[4] The idea for this generalization was suggested by Luc Segoufin.

position $i+1$ whose state coordinate is obtained from q by applying the letter a_{i+1}. A partial run is a set of configurations which forms a chain under the successor relation. Using this set notation we can talk about subsets of runs.

Below we define the data structure, show how it can be computed in time $O(|Q| \cdot |w|)$, and then how it can be used to answer infix queries in time $O(|Q|)$.

The data structure. The structure stores a set R partial runs, called *tapes*. Each tape is assigned a rank in $\{1, \ldots, |Q|\}$.

1. Each configuration appears in exactly one tape.
2. For any position i, the tapes that contain configurations on position i have pairwise different ranks.
3. Let (q, i) be a configuration appearing in tape $\rho \in R$. The tape of its successor configuration is either ρ or has smaller rank than ρ.

The data structure contains a record for each tape, which stores its rank as well as a pointer to its last configuration. Each configuration in the word stores a pointer to its tape, i.e. there is a two-dimensional array of pointers to tapes, indexed states q and by word positions i. We have a second two-dimensional array, indexed by word positions i and ranks j, which on position (i, j) stores the unique configuration on position i that belongs to a tape of rank j.

Computing the data structure. The data structure is constructed in a left-to-right pass through the word. Suppose we have calculated the data structure for a prefix $a_1 \cdots a_i$ and we want to extend it to the prefix $a_1 \cdots a_{i+1}$. We extend all the tapes that contain configurations for position i with their successor configurations. If two tapes collide by containing the same configuration on position $i+1$, then we keep the conflicting configuration only in the tape with smaller rank and remove it from the tape with larger rank. We start new tapes for all configurations on position $i+1$ that are not successors of configurations on position i, and assign to them ranks that have been freed due to collisions.

Using the data structure. Let (q, i) be a configuration. For a position $j \geq i$, let π be the run that begins in (q, i) and ends in position j. We claim that $O(|Q|)$ operations are enough to find the configuration from π on position j. How do we do this? We look at the last configuration (p, m) in the unique tape ρ that contains (q, i) (each tape has a pointer to its last configuration). If $m \geq j$, then $\rho \supseteq \pi$, so all we need to do is find the unique configuration on position j that belongs to a tape with the same rank as ρ (this will actually be the tape ρ). For this, we use the second two-dimensional array from the data structure. If $m < j$, we repeat the algorithm, by setting (q, i) to be the successor configuration of (p, m). This terminates in at most $|Q|$ steps, since each repetition of the algorithm uses a tape ρ of smaller rank.

Comments. After seeing the construction above, the reader may ask: what is the point of the factorization forest theorem, if it can be avoided, and the resulting construction is simpler and more efficient? There are two answers to this

question. The first answer is that there are other applications of factorization forests. The second answer is more disputable. It seems that the factorization forest theorem, like other algebraic results, gives an insight into the structure of regular languages. This insight exposes results, which can then be proved and simplified using other means, such as automata. To the author's knowledge, the algorithm from Theorem 2 came before the algorithm from Theorem 4, which, although straightforward, seems to be new.

3 Well-typed regular expressions

In this section, we use the Factorization Forest Theorem to get a stronger version of the Kleene theorem. In the stronger version, we produce a regular expression which, in a sense, respects the syntactic monoid of the language.

Let $\alpha : A^* \to S$ be a morphism. As usual, we write *type of* w for $\alpha(w)$. A regular expression E is called well-typed for α if for each of its subexpressions F (including E), all words generated by F have the same type.

Theorem 5. *Any language recognized by a morphism* $\alpha : A^* \to S$ *can be defined by a union of regular expression that are well-typed for* α.

Proof
By induction on k, we define for each $s \in S$ a regular expression $E_{s,k}$ generating all words of type s that have an α-factorization forest of height at most k:

$$E_{s,1} \quad := \bigcup_{\substack{a \in A \cup \{\epsilon\} \\ \alpha(a)=s}} a \qquad\qquad E_{s,k+1} \quad := \bigcup_{\substack{u,t \in S \\ ut=s}} E_{u,k} \cdot E_{t,k} \quad \underbrace{\cup \quad (E_{s,k})^+}_{\text{if } s = ss} .$$

Clearly each expression $E_{s,k}$ is well-typed for α. The Factorization Forests Theorem gives an upper bound K on the height of α-factorizations needed to get all words. The well-typed expression for a language $L \subseteq A^*$ recognized by α is the union of all expressions $E_{s,K}$ for $s \in \alpha(L)$. ∎

3.1 An effective characterization of $\Sigma_2(<)$

In this section, we use Theorem 5 to get an effective characterization for Σ_2. First, we explain what we mean by effective characterization and Σ_2.

Let $\mathcal{L}$ be a class of regular languages (such as the class of finite languages, or the class of star-free languages, etc.). We say $\mathcal{L}$ has an *effective characterization* if there is an algorithm, which decides if a given regular language L belongs to the class $\mathcal{L}$. As far as decidability is concerned, the representation of L is not important, here we use its syntactic morphism. There is a large body of research on effective characterizations of classes of regular languages. Results are difficult to obtain, but the payoff is often a deeper understanding of the class $\mathcal{L}$.

Often the class $\mathcal{L}$ is described in terms of a logic. A prominent example is first-order logic. The quantifiers in a formula range over word positions. The

signature contains a binary predicate $x < y$ for the order on word positions, and unary a predicate $a(x)$ for each letter $a \in A$ of the alphabet that tests the label of a position. For instance, the word property "the first position has label a" can be defined by the formula $\exists x \big(a(x) \wedge (\forall y\ y \geq x)\big)$. A theorem of McNaughton and Papert [11] says that first-order logic defines the same languages as star-free expressions, and Schützenberger [13] gives an effective characterization of the star-free languages (and therefore also of first-order logic).

A lot of attention has been devoted to the quantifier alternation hierarchy in first-order logic, where each level counts the alterations between $\forall$ and $\exists$ quantifiers in a first-order formula in prenex normal form. Formulas that have $n-1$ alternations (and therefore n blocks of quantifiers) are called Σ_n if they begin with $\exists$, and Π_n if they begin with $\forall$. For instance, the language "nonempty words with at most two positions that do not have label a" is defined by the Σ_2 formula

$$\exists x_1 \exists x_2 \forall y. \quad (y \neq x_1 \wedge y \neq x_2) \quad \Rightarrow \quad a(y)\ .$$

Effective characterizations are known for levels Σ_1 (a language has to be closed under adding letters), and similarly for Π_1 (the language has to be closed under removing letters). For languages that can be defined by a boolean combination of Σ_1 formulas, an effective characterization is given by Simon [14]. The last levels with a known characterization are Σ_2 and Π_2. For all higher finite levels, starting with boolean combinations of Σ_2, finding an effective characterization is an important open problem.

Below, we show how the well-typed expressions from Theorem 5 can be used to give an effective characterization of Σ_2. The idea to use the Factorization Forests Theorem to characterize Σ_2 first appeared in [12], but the proof below is based on [2]. Fix a regular language $L \subseteq A^*$. We say a word w *simulates* a word w' if the language L is closed under replacing w' with w. That is, $uw'v \in L$ implies $uwv \in L$ for any for any $u, v \in A^*$. Simulation is an asymmetric version of syntactic equivalence: two words are syntactically equivalent if and only if they simulate each other both ways.

Theorem 6. *Let $L \subseteq A^*$ be a regular language, and $\alpha : A^* \to S$ be its syntactic morphism. The language L can be defined in Σ_2 if and only if*

> *(*) For any words w_1, w_2, w_3 mapped by α to the same idempotent $e \in S$ and v a subsequence of w_2, the word $w_1 v w_3$ simulates $w_1 w_2 w_3$.*

Although it may not be immediately apparent, condition (*) can be decided when given the syntactic morphism of L. The idea is to calculate, using a fixpoint algorithm, for each $s, t \in S$ if some word of type s has a subsequence of type t.

The "only if" implication is done using a standard logical argument, and we omit it here. The more difficult "if" implication will follow from Lemma 2. The lemma uses *overapproximation*: we say a set of words K overapproximates a subset $K' \subseteq K$ if every word in K simulates some word in K'.

Lemma 2. *Assume (*). Any regular expression that is well-typed for α can be overapproximated by a language in Σ_2.*

Before proving the lemma, we show how it gives the "if" part in Theorem 6. Thanks to Theorem 5, the language L can be defined as a finite union of well-typed expressions. By Lemma 2, each of these can be overapproximated in Σ_2. The union of overapproximations gives exactly L: it clearly contains L, but contains no word outside L by definition of simulation.

Proof (of Lemma 2)
Induction on the size of the regular expression. The induction base is simple. In the induction step, we use closure of Σ_2 under union and concatenation.

Union in the induction step is simple: the union of overapproximations for E and F is an overapproximation of the union of E and F. For concatenation, we observe that simulation is compatible with concatenation: if w simulates w' and u simulates u', then wu simulates $w'u'$. Therefore, the concatenation of overapproximations for E and F is an overapproximation of $E \cdot F$.

The interesting case is when the expression is F^+. Since F is well typed, all words in F have type, say $e \in S$. Since F^+ is well-typed, e must be idempotent. Let M be an overapproximation of F obtained from the induction assumption. Let A_e be the set of all letters that appear in words of type e. As an overapproximation for F^+, we propose

$$K = \qquad M \quad \cup \quad M(A_e)^* M \ .$$

A Σ_2 formula for K can be easily obtained from a Σ_2 formula for M. Since every word in F is built from letters in A_e, we see that K contains F^+. To complete the proof of the lemma, we need to show that every word in K simulates some word in F^+. Let then w be a word in K. If w is in M, we use the induction assumption. Otherwise, w can be decomposed as $w = w_1 v w_3$, with $w_1, w_3 \in M$ and v a word using only letters from A_e. By induction assumption, w_1 simulates some word $w_1' \in F$ and w_3 simulates some word $w_3' \in F$. Since simulation is compatible with concatenation, $w_1 v w_3$ simulates $w_1' v w_3'$. Since e is idempotent, each word in $(A_e)^*$ is a subsequence of some word of type e. In particular, v is a subsequence of some word v' of type s. By condition (*), $w_1' v w_2'$ simulates $w_1' v' w_2' \in F^+$. The result follows by transitivity of simulation. ∎

Corollary 1 A language is definable in Σ_2 if and only if it is a union of languages of the form

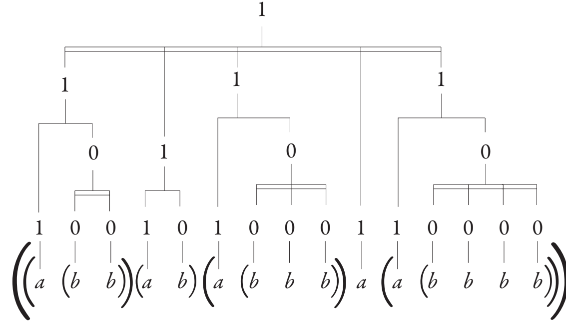
$$A_0^* a_1 A_1^* \cdots A_{n-1}^* a_n A_n^* \tag{2}$$

Proof
The "if" part is immediate, since each expression as in (2) can be described in Σ_2. The "only if" part follows by inspection of the proof of Lemma 2 where, instead of a formula of Σ_2, we could have just as well produced a union of languages as in (2). ∎

4 Transducers

The proof of the Factorization Forests Theorem also shows that factorization forests can be computed in linear time. In this section we strengthen that statement by showing that factorization forests can be produced by transducers.

A tree can be written as a word with matched parentheses. This idea can be applied to factorizations, as shown by the following picture:



To aid reading, we have removed the parentheses around individual letters (which correspond to factorization forests of height 1).

We can therefore define the word encoding of a factorization as a word over an extended alphabet $A \cup \{(,)\}$ that also contains an opening parenthesis, and a closing one. We write w_f for the word encoding of a factorization f. The following lemma shows that factorizations can be calculated by a transducer.

Lemma 3. *Fix a morphism $\alpha : A^* \to S$ and a height $k \in \mathbb{N}$. There is a nondeterministic transducer $\mathcal{T}_k : A^* \to (A \cup \{(,)\})^*$, which produces on input $w \in A^*$ the word encodings of all α-factorizations of w of height at most k.*

Proof
Induction on k. ∎

There are two problems with the transducer $\mathcal{T}_k$.

The first is nondeterminism. For instance, we might want to use the transducer to find a factorization forest, and nondeterminism seems to gets in the way. This particular problem with nondeterminism can be dealt with: as for any nondeterministic transducer, one can compute (some) output in $\mathcal{T}_k(w)$ in time proportional to the length of w times the number of states in $\mathcal{T}_k$. (In particular, assuming that the morphism α is fixed, we get a linear time algorithm for computing an α-factorization.) However, nondeterminism turns out to be a serious problem for applications to tree languages, as we will see later.

A second problem is that $\mathcal{T}_k$ has a lot of states. This is because the construction of $\mathcal{T}_k$, at least the easy inductive construction suggested above, gives a state space that is exponential in k.

A nice solution to this problem was proposed by Thomas Colcombet. He shows that if the conditions on a factorization forest are relaxed slightly, then the factorization can be output by a deterministic transducer with $O(|S|)$ states.

What is the relaxation on factorizations? Recall the idempotent rule, which allowed to split a word w into $w = w_1 \cdots w_n$ as long as all the factors $w_1, \ldots, w_n$ had the same idempotent type. This requirement could be stated as

$$\alpha(w_i) \cdot \alpha(w_j) = \alpha(w_j) \cdot \alpha(w_i) = \alpha(w_i) \qquad \text{for all } i, j \in \{1, \ldots, n\}.$$

In other words, the type of any word w_i absorbs the type of any other word w_j, both on the left and on the right. In [5, 6] Colcombet proposed a relaxed version of this rule, where the type only absorbs to the right:

$$\alpha(w_i) \cdot \alpha(w_j) = \alpha(w_i) \qquad \text{for all } i, j \in \{2, \ldots, n-1\}.$$

We will use the term *forward Ramseyan rule* for a rule that allows a split $w = w_1 \cdots w_n$ under the above condition. A factorization that uses the forward Ramseyan rule instead of the idempotent rule is called a *forward Ramseyan factorization*. Every factorization that uses the idempotent rule is a forward Ramseyan factorization (since the condition in the forward Ramseyan rule is weaker than the condition in the idempotent rule), but not vice versa.

Despite being more relaxed, in most cases the forward Ramseyan rules gives the same results as the idempotent rule. Consider, for example, the infix problem from Theorem 2. Suppose we have a word split $w = w_1 \cdots w_n$ according the forward Ramseyan rule, and that we know the values $\alpha(w_1), \ldots, \alpha(w_n)$. Suppose that we want to calculate the type $\alpha(w_i \cdots w_j)$ for some $i \leq j \in \{2, \ldots, n-1\}$. Thanks to the forward Ramseyan rule, this type is

$$\alpha(w_i \cdots w_j) = \alpha(w_i)\alpha(w_{i+1})\alpha(w_{i+2} \cdots w_j) = \alpha(w_i)\alpha(w_{i+2} \cdots w_j) = \cdots = \alpha(w_i) \ .$$

If we are interested in the case of $i = 1$ (a similar argument works for $j = n$), then we first find the type $\alpha(w_2 \cdots w_j)$ and then prepend the type of $\alpha(w_1)$.

The reason why Colcombet introduced forward Ramseyan factorizations is that they can be produced by a deterministic transducer (we use the same encoding of factorizations as words over the alphabet A_S).

Theorem 7 (Colcombet [5, 6]). *Fix a morphism $\alpha : A^* \to S$. There is a deterministic transducer $\mathcal{T}_k : A^* \to (A \cup \{(,)\})^*$, which produces, on input $w \in A^*$, the word encoding of a forward Ramseyan factorization of w of height at most $|S|$.*

We cite below two applications of this result. The first concerns trees, and the second concerns infinite words.

Trees. Suppose we have a tree, and we want to calculate factorizations for words that label paths in the tree. There are two difficulties, both related to the fact that paths have common prefixes, as in the picture below:

.

The first difficulty is that the combined length of all paths in the tree can be quadratic in the number of nodes. The second difficulty is that the factorizations for two different paths may be inconsistent on their common prefixes. Both of these difficulties are solved by using the deterministic transducer from Theorem 7, and running it on each path, from root to leaf. Along these lines, Theorem 7 was used in [4] to provide a linear time algorithm for evaluating XPath queries on XML documents.

Infinite words. The transducer in Theorem 7 can also be used on an infinite word $w = a_1 a_2 \cdots$. It also produces a forward Ramseyan factorization. The only difference is that after some point, we will start to see an infinite sequence of matched parentheses $(..)(..)(..)\cdots$ at the same nesting level (some of the initial parentheses might remain open forever). This construction has been used in [6] to determinize automata on infinite words (that is, convert a Büchi automaton into an equivalent Muller automaton).

5 Limitedness

In this last section, we talk about limitedness of automata. This is the original setting in which the Factorization Forests Theorem were used, so a discussion of the theorem would be incomplete without mentioning limitedness. On the other hand, the subject is quite technical (but fascinating), so we only sketch the problem, and point the reader to the literature.

A *distance automaton* is a nondeterministic automaton where a subset of the states is declared costly. The *cost* of a run ρ is the number of times it uses the costly states. The cost of a word $w \in A^*$ is the minimal cost of a run (from an initial to a finite state) over this word. If there is no run, the cost is 0. The automaton is called *limited* if there is a finite bound on the cost of all words.

We want an algorithm that decides if a distance automaton is limited. In other words, we want to decide if the expression

$$\max_{w \in A^*} \quad \min_{\rho \in \text{runs}(w)} \quad \text{cost}(\rho)$$

has a finite value. The difficulty of the problem comes from the alternation between max and min. If the expression had been max max, the problem could

be decided by simply searching for a loop in the automaton that uses a costly state. (In particular, the limitedness problem is straightforward for deterministic automata.) If the expression had been $\min \min$ or $\min \max$, the problem would trivialize, since the value would necessarily be finite.

The limitedness problem is closely related to star height. The star height of a regular expression is the nesting depth of the Kleene star. For instance, the expression $a^* + b^*$ has star height 1, while the expression $((a + b)^* aa)^*$ has star height 2, although it is equivalent to $(a + b)^* aa$, which has star height 1. Complementation is not allowed in the expressions (when complementation is allowed, we are talking about generalized star height). The star height problem is to decide, given a regular language L and a number k, if there exists an expression of star height k that defines L. This famous problem has been solved by Hashiguchi [7]. An important technique in the star height problem is limitedness of distance automata. Distance automata have been introduced by Hashiguchi, and the limitedness problem was studied by Leung [10] and Simon [16]. The latter paper is the first important application of the Factorization Forests Theorem.

The current state of the art in the star height problem is the approach of Daniel Kirsten [8], who uses an extension of distance automata. The extended model is called a distance desert automaton, and it extends a distance automaton in two ways. First, a distance desert automaton keeps track of several costs (i.e. if the cost is seen as the value of a counter, then there are several counters). Second, the cost can be reset, and the cost of a run is the maximal cost seen at any point during the run. The star height problem can be reduced to limitedness of distance desert automata: for each regular language L and number k, one can write a distance desert automaton that is limited if and only if the language L admits an expression of star height k. In [8], Daniel Kirsten shows how to decide limitedness for distance desert automata, and thus provides another decidability proof for the star height problem.

A related line of work was pursued in [3]. This paper considered a type of distance desert automaton (under the name BS-automaton), which would be executed on an infinite word. (The same type of automata was also considered in [1], this time under the name of R-automata.) The acceptance condition in a BS-automaton talks about the asymptotic values of the cost in the run, e.g. one can write an automaton that accepts infinite words where the cost is unbounded. The main contribution in [3] is a complementation result. This complementation result depends crucially on the Factorization Forests Theorem.

References

1. P. A. Abdulla, P. Krcál, and W. Yi. R-automata. In *CONCUR*, pages 67–81, 2008.
2. M. Bojańczyk. The common fragment of ACTL and LTL. In *Foundations of Software Science and Computation Structures*, pages 172–185, 2008.
3. M. Bojańczyk and T. Colcombet. Omega-regular expressions with bounds. In *Logic in Computer Science*, pages 285–296, 2006.

4. M. Bojanczyk and P. Parys. XPath evaluation in linear time. In *PODS*, pages 241–250, 2008.
5. T. Colcombet. A combinatorial theorem for trees. In *ICALP'07*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
6. T. Colcombet. Factorisation forests for infinite words. In *FCT'07*, 2007.
7. K. Hashiguchi. Algorithms for determining relative star height and star height. *Inf. Comput.*, 78(2):124–169, 1988.
8. D. Kirsten. Distance desert automata and the star height problem. *Theoretical Informatics and Applications*, 39(3):455–511, 2005.
9. Manfred Kufleitner. The height of factorization forests. In *MFCS*, pages 443–454, 2008.
10. Hing Leung. The topological approach to the limitedness problem on distance automata. *Idempotency*, pages 88–111, 1998.
11. R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, Cambridge Mass., 1971.
12. J.-É. Pin and P. Weil. Polynomial closure and unambiguous product. *Theory Comput. Systems*, 30:1–30, 1997.
13. M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, 1965.
14. I. Simon. Piecewise testable events. In *Automata Theory and Formal Languages*, pages 214–222, 1975.
15. I. Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72:65–94, 1990.
16. Imre Simon. On semigroups of matrices over the tropical semiring. *ITA*, 28(3-4):277–294, 1994.