View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIR Universita degli studi di Milano

DIPARTIMENTO DI SCIENZE DELL'INFORMAZIONE

Rapporto interno N. 323 - 08

Integer compositions and syntactic trees of repeat-until programs

Luca Breveglieri, Stefano Crespi Reghizzi,
Massimiliano Goldwurm

Integer compositions and syntactic trees of repeat-until programs

Luca Breveglieri[(1)] Stefano Crespi Reghizzi[(1)]

Massimiliano Goldwurm[(2)]

(1) Dipartimento di Elettronica e Informazione, Politecnico di Milano, via Ponzio 34/5, 20133 Milano – Italy
{luca.breveglieri, stefano.crespireghizzi}@polimi.it

(2) Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
Via Comelico 39-41, 20135 Milano – Italy, goldwurm@dsi.unimi.it

Rapporto Interno
RI - DSI n. 323 - 08

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39/41, 20135 Milano, Italy

Marzo 2008

Abstract

In this work we study some properties of integer compositions in connection with the recognition of rational trace languages. In particular, we introduce some operations defined on integer compositions and present procedures for their computation that work in linear or in quadratic time. These procedures turn out to be useful in the analysis of syntactic trees of certain regular expressions, called repeat-until expressions, which intuitively represent programs of instructions nested in repeat-until loops. Our main aim is to show how, in some cases, such an analysis allows us to design algorithms for the recognition of (rational) trace languages defined by repeat-until expressions, which work in quadratic time independently of the concurrency relation.

Keywords: Automata and Formal Languages, Trace Languages.

1 Introduction

The recognition of trace languages is a classical problem widely studied in the literature [4, 10, 6, 1]. In the rational case the problem can be defined as follows: given an independence alphabet (Σ, I) and a regular language $L \subseteq \Sigma^*$, one has to verify for an input $x \in \Sigma^+$ whether $[x]_I \cap L \neq \emptyset$, i.e. whether the trace $[x]_I$ belongs to the trace language $[L]_I$ generated by L. It is well-known that the problem can be solve in time $O(n^\alpha)$, where $n = |x|$ and α is the size of the maximum clique in (Σ, I) [4, 6]. Moreover, the uniform version of the problem, where both (Σ, I) and a description of L are part of the input, becomes Np-complete. Another algorithm is given in [1] which depends on the prefixes of the input trace, and hence a probabilistic analysis of the procedure is obtained assuming equiprobable all input strings of given length.

In this work we study the problem in the case when the language L is defined by a repeat-until expression, i.e. a regular expression over Σ that includes only concatenation and $^+$ operation and where each $a \in \Sigma$ occurs only once. An expression α of this type represents a program of nested repeat-until loops, where Σ is the set of instructions, and the language L can be seen as the set of executions of the program. Thus, given an independence relation I over Σ, the recognition of the trace language $[L]_I$ is equivalent to verifying whether a sequence of instructions given in input can be rearranged according to I to become an effective execution of the program α.

It turns out that the words of any language defined by a repeat-until expression admit a syntactic tree that can be easily represented by integer compositions. For this reason we study some properties of the integer compositions and in particular we introduce specific operations on such structures that can be computed in linear or in quadratic time (with respect of the input compositions). These results can be used to design algorithms for the recognition of rational trace languages represented by string languages defined by repeat-until expressions. We show an empirical method that can produce algorithms for this problem working in $O(n^2)$ time, independently of the concurrency relation. A drawback of the present contribution is that such a method is not general and can yield an effective procedure only in some cases, intuitively when the construction of the associated syntactic tree (by means of the above operations on integer compositions) does not yield ambiguities.

The material we present is organized as follows. In Section 3 we introduce an algebra to manipulate integer compositions, based on operations of product, quotient, matching, contraction, expansion, and describe efficient algorithms for their computation. Repeat-until expressions are introduced in Section 4 together with a notion of syntactic tree for the words of any language defined by such an expression. In Section 5 we show how one can design quadratic time algorithms for the recognition of rational trace languages defined by certain repeat-until expressions.

2 Basic notions

Given a finite alphabet Σ and a word $x \in \Sigma^*$, $|x|$ represents the length of x while, for each $a \in \Sigma$, $|x|_a$ is the number of occurrences of a in x. More generally, for a word $y \in \Sigma^+$, $|x|_y$ denotes the number of occurrences of y in x. Moreover, given a subset $A \subseteq \Sigma$, $\pi_A(x)$ is the projection of x over A. Further, if x is not the empty word ε, $P(x)$ and $U(x)$ denote, respectively, the first and the last symbol of x, while $S_1(x)$ is the suffix of x of length $|x| - 1$.

Given a word $x \in \{a, b\}^*$, a *run* of a in x is an occurrence of a maximal factor of x included in $\{a\}^+$. An analogous definition holds for b. For instance, the word *aaabbabbbaaa* has 3 runs of a and 2 runs of b (*aaa*, *a*, *aaa* and *bb*, *bbb*, respectively). Clearly, two words $x, y \in \{a, b\}^+$ are equal if they have the same sequence of runs of a, the same sequence of runs of b and $P(x) = P(y)$.

There is a natural relationship between runs of a letter in binary words and compositions of integers. A *composition* of an integer $n \geq 1$ is a nonempty finite sequence $(i_1, i_2, \ldots, i_h)$ of integers such that $i_j \geq 1$ for every $j = 1, \ldots, h$ and $\sum_{j=1}^{h} i_j = n$ (see for instance [8]). Thus, every word $x \in \{a,b\}^+$, where $a \neq b$, $|x|_a \geq 1$ and $|x|_b \geq 1$, defines two compositions γ_a and γ_b determined respectively by the runs of a and the runs of b in x. More precisely, $\gamma_a = (i_1, i_2, \ldots, i_h)$ is a composition of $|x|_a$, where h is the number of runs of a in x and each i_j is the length of the j-th run. Analogously, γ_b is a composition of $|x|_b$ defined in a similar way. We also say that γ_a (resp., γ_b) is the composition *generated* by x on a (resp., b).

Now, let us recall some basic definitions on traces. An independence relation I on Σ is a binary relation on Σ, i.e. $I \subseteq \Sigma \times \Sigma$, that is irreflexive and symmetric. For every $a, b \in \Sigma$ we say that a and b are independent if $(a,b) \in I$ and in this case we also write aIb. The dependence relation D is the complement of I, that is $D = \{(a,b) \in \Sigma \times \Sigma \mid (a,b) \notin I\}$. We say that a and b are dependent if $(a,b) \in D$ and also in this case we write aDb. An independence relation I establishes an equivalence relation $\equiv_I$ on Σ^* as the reflexive and transitive closure of the relation $\sim_I$ defined by

$$xaby \sim_I xbay \qquad \forall x, y \in \Sigma^*,\ \forall (a,b) \in I.$$

The relation $\equiv_I$ is a congruence over Σ^*, i.e. an equivalence relation preserving concatenation between words. For every $x \in \Sigma^*$ the equivalence class $[x] = \{y \in \Sigma^* \mid y \equiv_I x\}$ is called trace, the quotient monoid $\Sigma^* / \equiv_I$ is called trace monoid and usually denoted by $M(\Sigma, I)$. The pair (Σ, I) is called independence alphabet and it is usually represented by an undirected graph where Σ is the set of nodes and I the set of edges. For every trace monoid $M(\Sigma, I)$ the subsets $T \subseteq M(\Sigma, I)$ are called trace languages and, for every $L \subseteq \Sigma^*$, we define $[L] = \{[x] \in M(\Sigma, I) \mid x \in L\}$ as the trace language represented by L. A trace language is called rational if it is represented by a regular language. The class of rational trace languages has been widely studied in the literature and it coincides with the smallest family of trace languages including the finite sets in $M(\Sigma, I)$ and closed under the operation of union, product and Kleene closure (over the trace monoid).

Here we are particularly interested in the recognition problem of rational trace languages. For a given independence alphabet (Σ, I) and a given regular language $L \subseteq \Sigma^*$, such a problem consists of verifying, for an input $x \in \Sigma^*$, whether $[x] \in [L]$, that is whether there exists a word $w \in [x]$ belonging to L.

3 Algebra of compositions

In this section we study some properties of the integer compositions. Our purpose is to present some operations on such structures and describe the algorithms for their computation.

We recall that a *composition* of an integer $n \geq 1$ is a nonempty finite sequence $(i_1, i_2, \ldots, i_h)$ of integers such that $i_j \geq 1$ for every $j = 1, \ldots, h$ and $\sum_{j=1}^{h} i_j = n$. Integer compositions are classical combinatorial structures. For instance it is well-known that there are 2^{n-1} compositions of any integer $n \geq 1$ [8]. A natural notion associated with such structures is the inclusion relation among compositions of the same integer, that we denote by $\preceq$.

Definition 1 *Given two compositions $\alpha = (a_1, a_2, \ldots, a_h)$ and $\beta = (b_1, b_2, \ldots, b_m)$ of an integer $n \geq 1$, we say that α is* finer *than β (or β is* coarser *than α), and write*

$$\alpha \preceq \beta$$

if $h \geq m$ and there are m indices $\ell_1, \ell_2, \ldots, \ell_m$ such that $1 \leq \ell_1 < \ell_2 < \cdots < \ell_m = h$ and

$$b_1 = \sum_{j=1}^{\ell_1} a_j\ , \quad b_2 = \sum_{j=\ell_1+1}^{\ell_2} a_j\ , \ \ldots\ , \quad b_m = \sum_{j=\ell_{m-1}+1}^{\ell_m} a_j$$

Note that if $\alpha \preceq \beta$ then there exists a unique m-tuple of indices $\ell_1, \dots, \ell_m$ satisfying the previous property. Moreover, $\preceq$ is a partial order relation on the family of all compositions of n, where $(1,1,\dots,1)$ is the smallest element and (n) the largest one.

Clearly, there are $O(n)$ time algorithms that on input α, β verify whether $\alpha \preceq \beta$ and, in the affirmative case, compute the corresponding sequence $\ell_1, \dots, \ell_m$ defined above.

In the following, we often represent a composition $\alpha = (a_1, a_2, \dots, a_h)$ in the form $\alpha = (a_i)_h$ and denote by n_α the corresponding integer, i.e. $n_\alpha = \sum_{i=1}^{h} a_i$.

3.1 Product operation

The product is our simplest operation between compositions and is defined as follows.

Definition 2 *Consider two compositions $\alpha = (a_i)_h$ and $\beta = (b_j)_k$, and assume $n_\alpha = k$, which implies $k \geq h$. Then, the product $\alpha \cdot \beta$ is the composition $\gamma = (g_l)_h$ such that*

$$g_l = \sum_{j=j_{l-1}+1}^{j_l} b_j \qquad \text{for every} \quad l = 1,2\dots,h$$

where $j_0 = 0$ and $j_l = \sum_{i=1}^{l} a_i$ for each $l = 1,2\dots,h$.

More precisely, we have

$$\begin{array}{rcl} g_1 & = & b_1 + b_2 + \cdots + b_{a_1} \\ g_2 & = & b_{a_1+1} + b_{a_1+2} + \cdots + b_{a_1+a_2} \\ \dots & = & \dots \\ g_h & = & b_{a_1+\cdots+a_{h-1}+1} + b_{a_1+\cdots+a_{h-1}+2} + \cdots + b_{a_1+\cdots+a_h} \end{array}$$

Briefly, γ is obtained from β by adding consecutive elements as indexed by the composition α. Clearly, we have $\beta \preceq \gamma$ and $n_\gamma = n_\beta$.

Here is an example:

$$\alpha = (1,2,2)_3 \qquad \beta = (1,2,1,3,2)_5 \qquad \gamma = \alpha \cdot \beta = (1,3,5)_3$$

Notice that in general the product is not commutative. Moreover, for every composition $\beta = (b_j)_k$, the following identities hold:

$$(1,1,\dots,1)_k \cdot \beta = \beta \qquad (k)_1 \cdot \beta = \left(n_\beta\right)_1 \qquad \beta \cdot (1,1,\dots,1)_{n_\beta} = \beta$$

The product of two compositions can be computed by scanning their elements from left to right. Here is an algorithm for computing the product of two compositions $\alpha = (a_i)_h$, $\beta = (b_j)_k$ such that $n_\alpha = k$:

Algorithm - Product of compositions

input α, β
$l = 1$
for $i = 1$ **to** h **do**
  $g_i = 0$
  for $j = 1$ **to** a_i **do**
    $g_i = g_i + b_l$
    $l = l + 1$
  end for
  output g_i
end for

The algorithm outputs the elements of the product composition. Clearly it has a linear time complexity $O(n)$, where $n = n_\alpha = k$.

3.2 Quotient operation

If two compositions are related by the partial order $\preceq$, it is possible to define a quotient operation between them.

Definition 3 *Given two compositions $\alpha = (a_i)_h$, $\beta = (b_j)_k$ where $\alpha \preceq \beta$ (and hence $k \leq h$), consider the sequence of indices $\ell_0, \ell_1, \ldots \ell_k$ such that $0 = \ell_0 < \ell_1 < \cdots < \ell_k = h$ and*

$$b_j = \sum_{i=\ell_{j-1}+1}^{\ell_j} a_i \qquad \textit{for every } j = 1, 2, \ldots, k$$

Then, the quotient β/α is the composition $\gamma = (g_j)_k$ of h such that

$$g_j = \ell_j - \ell_{j-1} \qquad \textit{for every } j = 1, 2, \ldots, k$$

Intuitively, the quotient operation creates a new composition γ representing the partition of elements of α to be added up in order to get β. It is clear that $\gamma = \beta/\alpha$ implies $\beta = \gamma \cdot \alpha$.

For instance:

$$\beta = (4,2,5)_3 \qquad \alpha = (1,3,2,1,1,3)_6 \qquad \gamma = \beta/\alpha = (2,1,3)_3$$

Notice that we have the following special cases, for any composition $\alpha = (a_i)_h$:

$$\alpha/\alpha = (1,1,\ldots,1)_h \qquad (n_\alpha)_1/\alpha = (h)_1 \qquad \alpha/(1,1,\ldots,1)_{n_\alpha} = \alpha$$

Also the quotient of two compositions can be computed in linear time by scanning both operands from left to right. Here is an algorithm that, for an input $\alpha = (a_i)_h$, $\beta = (b_j)_k$ satisfying the relation $\alpha \preceq \beta$, computes the composition $\gamma = (g_j)_k$ such that $\gamma = \beta/\alpha$.

Algorithm - Quotient of compositions

```
input α, β
i = 1
for j = 1 to k do
    g_j = 0
    s = 0
    while s + a_i ≤ b_j do
        g_j = g_j + 1
        s = s + a_i
        i = i + 1
    end while
    output g_j
end for
```

This algorithm outputs the elements of the quotient composition and has a linear time complexity $O(h)$.

3.3 Matching operation

We have seen that the quotient operation is the inverse of the product, in the sense that $\alpha \cdot \beta = \gamma$ implies $\alpha = \gamma/\beta$. Here we introduce another operation, which allows us to determine β from γ and α. The main difference with respect to the previous operations is that now the result is not unique.

Formally, given two compositions $\alpha = (a_i)_h$ and $\beta = (b_i)_h$, where $a_i \leq b_i$ for each $i = 1, \ldots, h$, a matching of α and β is a composition $\delta = (d_j)_{n_\alpha}$ such that $\alpha \cdot \delta = \beta$, that is setting $0 = \ell_0$, $\ell_1 = a_1$, $\ell_2 = a_1 + a_2$, ..., $\ell_h = n_\alpha$ we have

$$b_i = \sum_{j=\ell_{i-1}+1}^{\ell_i} d_j \qquad \text{for every } i = 1, \ldots, h \, .$$

Observe that $\delta \preceq \beta$. Moreover, δ may not be unique since its elements are obtained from possible different decompositions of the b_j's. We denote by $\beta \oslash \alpha$ the set of all matchings of α and β.

As an example, let $\alpha = (1,2,3)_3$ and $\beta = (1,3,3)_3$. Then,

$$\beta \oslash \alpha = \{(1,1,2,1,1,1)_6, (1,2,1,1,1,1)_6\}$$

Observe that a matching of $\alpha = (a_i)_h$ and $\beta = (b_j)_k$ always exists whenever $h = k$ and $a_i \leq b_i$ for every i; the matching is unique if $\alpha = \beta$ and in this case it coincides with $(1,1,\ldots,1)_{n_\alpha}$. It is also clear that computing a matching $\delta \in \beta \oslash \alpha$ (if any) can be done in time $O(n_\alpha)$.

3.4 Contraction of compositions

Here we study a another operation on compositions, called contraction, that is again partial and when defined it may yield more than one result.

Definition 4 *Consider two compositions $\alpha = (a_i)_h$, $\beta = (b_j)_k$ such that $h \geq k$ and $n_\alpha \leq n_\beta$. We say that a composition $\alpha' = (a'_j)_k$ is a* contraction *of α over β if the following conditions hold:*

$$\alpha \preceq \alpha' \tag{1}$$

$$a'_i \leq b_i \qquad \text{for all } i = 1, 2, \ldots, k. \tag{2}$$

Note that condition (1) implies $n_{\alpha'} = n_\alpha$. It is clear that there may be no contraction of two compositions: for instance this occurs when the maximum element of α is greater than any element of β. On the contrary, there may be more than one contraction of two compositions; as an example, the contractions of $\alpha = (1,2,1,1)$ over $\beta = (5,4)$ are the following compositions:

$$(1,4) \qquad (3,2) \qquad (4,1)$$

Also observe that there exists at most one contraction whenever $h = k$, i.e. α and β have the same length.

Now, let us define an algorithm that receives as input two compositions $\alpha = (a_i)_h$, $\beta = (b_j)_k$ such that $h < k$ and $n_\alpha \leq n_\beta$, it verifies whether there exists a contraction of α over β and, in the affirmative case, it effectively computes such a contraction α'. Observe that we evoid the case $h = k$, since this is reduced to check whether $a_i \leq b_i$ for each index i.

To solve the problem we compute a k-tuple $S_1, S_2, \ldots, S_k$ where intuitively, each S_i is a set of possible candidates for a'_i defined by pairs of indices (j, ℓ), $j \leq \ell$, such that $a'_i = a_j + \cdots + a_\ell$. More formally,

every S_i is a set of pairs $(j,\ell) \in \mathbb{N}^2$ where $1 \leq j \leq \ell \leq h$, defined as follows:

$$
\begin{aligned}
S_1 &= \left\{ (1,\ell) \in \mathbb{N}^2 \mid \sum_{t=1}^{\ell} a_t \leq b_1\,,\ k-1 \leq h-\ell \right\} \\
S_2 &= \left\{ (j,\ell) \in \mathbb{N}^2 \mid \sum_{t=j}^{\ell} a_t \leq b_2\,,\ \exists (s,j-1) \in S_1 \text{ for some } s \in \mathbb{N},\ k-2 \leq h-\ell \right\} \\
\dots & \quad \dots \\
S_i &= \left\{ (j,\ell) \in \mathbb{N}^2 \mid \sum_{t=j}^{\ell} a_t \leq b_i\,,\ \exists (s,j-1) \in S_{i-1} \text{ for some } s \in \mathbb{N},\ k-i \leq h-\ell \right\} \\
\dots & \quad \dots \\
S_k &= \left\{ (j,\ell) \in \mathbb{N}^2 \mid \sum_{t=j}^{\ell} a_t \leq b_k\,,\ \exists (s,j-1) \in S_{k-1} \text{ for some } s \in \mathbb{N},\ \ell \leq h \right\}
\end{aligned}
$$

Clearly, it may occur that S_i is empty for some i: in this case all subsequent S_j's (with $j > i$) are empty and there is no composition α' satisfying (1) and (2).

The following procedure computes all S_i's. Here, for a given i, *Init* is the set of indices j such that some pair (j,ℓ) belongs to S_i. Analogously, *Next* is the set of indices ℓ such that some pair $(j,\ell-1)$ belongs to S_i.

```
begin
    Init := {1}
    for i = 1,2,...,k do
        begin
            Next := ∅
            S_i := ∅
            for j ∈ Init do
                begin
                    ℓ := j
                    x := a_ℓ
                    while x ≤ b_i ∧ k − i ≤ h − ℓ do { add (j,ℓ) to S_i
                                                      ℓ := ℓ + 1
                                                      if ℓ ≤ h then { x := x + a_ℓ
                                                                      Next := Next ∪ {ℓ} }
                end
            if S_i = ∅ then { return no
                              stop }
            Init := Next
        end
end
```

Once the sequence $S_1, S_2, \dots, S_k$ is built, we look for a pair $(j,\ell) \in S_k$ such that $\ell = h$. If such a pair does not exist then there is no contraction of α over β. Otherwise, such a contraction α' can be computed by the following procedure, which builds a path backwards among the elements of $S_1, S_2, \dots, S_k$.

```
begin
```

```
        choose an element (j,t) ∈ S_k such that t = h
        compute a'_k = a_j + a_{j+1} + ··· + a_h
        for i = k−1, k−2, ..., 1 do
           begin
                find in S_i an element (r,ℓ) such that ℓ = j−1
                compute a'_i = a_r + a_{r+1} + ··· + a_ℓ
                j := r
           end
        return (a'_1, a'_2, ..., a'_k)
end
```

Let us evaluate the time complexity required by the first procedure. Checking whether $\ell \in Next$ can be done in constant time by using an array to implement the set $Next$. Thus the inner loop requires $O(1)$ time. Then, since the procedure executes three nested loops, it works in $O(kh^2)$ time.

Concerning the second procedure one can represent each S_i as an array of h lists $S_i(\ell)$, $\ell = 1,\ldots,h$, where every $S_i(\ell)$ contains the elements of the form (j,ℓ) in S_i. Then, searching for (j,t) in S_i with $t = \ell$ can be done by choosing the first element of $S_i(\ell)$, which requires constant time. As a consequence the second procedure takes $O(h)$ time.

As far as the space complexity is concerned, assume to use the above array representation for each set S_i. Then, in order to run the second procedure, we only need to maintain the first element of each $S_i(\ell)$ (if any). This allows us to implement the first procedure by using only $O(kh)$ space.

Computing contractions in quadratic time

The previous computation can be improved by using an algorithm that solves the problem in $O(h^2)$ time and space. Here, we describe in detail such a procedure.

As a first task, we compute all coefficients A_{ij}, for integers $1 \le i \le j \le h$, such that

$$A_{ij} = \sum_{t=i}^{j} a_t$$

This requires $O(h^2)$ time, since any A_{ij} with $i < j$ can be obtained from $A_{i\,j-1}$ by adding a_j.

In a second phase the algorithm computes, for every $i = 1,\ldots,k$, a family of pairs (j,ℓ), where $1 \le j \le \ell \le h$, such that $A_{j\ell} \le b_i$; thus $A_{j\ell}$ is a possible candidate for a'_i. The computation actually fills up a table $S = \{S_{i\ell} \mid i = 1,\ldots,k,\ \ell = 1,\ldots,h\}$, where each entry $S_{i\ell}$ equals the smallest index j such that $A_{j\ell} \le b_i$ and $S_{i-1\,j-1} \ne 0$ ($S_{i\ell}$ is set to 0 if such an index does not exist). More precisely, the entries of S are defined as follows:

i) For every $\ell = 1,\ldots,h$,

$$S_{1\ell} = \begin{cases} 1 & \text{if } A_{1\ell} \le b_1 \\ 0 & \text{otherwise} \end{cases}$$

ii) For any $i = 2,\ldots,k$ and every $\ell = 2,\ldots,h$, setting

$$T = \{j \in \mathbb{N} \mid 1 \le j \le \ell,\ A_{j\ell} \le b_i,\ S_{i-1\,j-1} \ne 0\},$$

we have

$$S_{i\ell} = \begin{cases} \min\{j \in T\} & \text{if } T \ne \emptyset \\ 0 & \text{otherwise} \end{cases}$$

To fill up S one can proceed row by row. For each $i = 1, \ldots, k$, the i-th row can be computed by using a list *Init* of initial indices j such that $S_{i-1\, j-1} > 0$. The elements of *Init* are maintained in increasing order and at the beginning *Init* only contains 1. During the computation, another list *New* is determined which contains the initial indices for row $i+1$ (this is done by adding $\ell + 1$ to *New* whenever $S_{i\ell}$ is set to a positive value). Clearly, if *New* remains empty for some row $i < k$ then all entries of the subsequent rows will only contain 0.

The details of the computation are described in the following procedure, where we assume that initially all entries of S are set to 0. Here, Λ denotes the empty list, $first(Init)$ is the first element of *Init* (which is *null* if $Init = \Lambda$) and j scans *Init* from the first to the last element.

```
begin
    Init := (1)
    for i = 1,2,...,k do
      begin
          New := Λ
          j := first(Init)
          ℓ := j
          while  j ≠ null ∧ ℓ ≤ h do
              begin
                  if ℓ < j then ℓ := j
                                    ⎧ S_iℓ := j
                  if  A_jℓ ≤ b_i then ⎨ ℓ := ℓ+1
                                    ⎩ if ℓ ≤ h then add ℓ to New
                          else j := next(j)
              end
          Init := New
      end
end
```

This procedure consists of two main loops. The outer one is iterated k many times, once for every $i = 1, \ldots, k$. For each value of i, the inner loop is repeated at most $2(h - i + 1)$ many times, once for every possible value of $j + \ell$. Since each iteration requires $O(1)$ time, the procedure works in $O(kh)$ time.

Once table S is filled in, it is easy to see that a composition α' satisfying (1) and (2) exists if and only if $S_{kh} > 0$. In this case, we can compute the integers a'_i, for $i = 1, \ldots, k$, by building a path backwards throughout the rows of S. The computation is described by the following procedure, which clearly works in $O(k)$ time.

```
begin
    j := S_kh
    a'_k := A_jh
    for i = k-1, k-2, ..., 1 do
      begin
          ℓ = j - 1
          j := S_iℓ
          a'_i := A_jℓ
      end
    return (a'_1, a'_2, ..., a'_k)
```

```
end
```

Observe that the computation of the coefficients A_{ij}, for $1 \leq i \leq j \leq h$, is the most expensive task in the algorithm. Hence the overal time and space complexity is $O(h^2)$.

3.5 Expansions of compositions

Now, let us consider a sort of dual version of the previous operation. Also in this case the operation is partial and may yield multiple results.

Definition 5 *Consider two compositions $\alpha = (a_i)_h$, $\beta = (b_j)_k$ such that $h \leq k$ and $n_\alpha \leq n_\beta$. We say that a composition $\alpha' = (a'_j)_k$ is a* expansion *of α over β if the following conditions hold:*

$$\alpha' \preceq \alpha \tag{3}$$

$$a'_i \leq b_i \qquad \text{for all } i = 1, 2, \ldots, k. \tag{4}$$

Of course condition (3) implies $n_{\alpha'} = n_\alpha$. It is also clear that there may be no expansion of α over β: this occurs for instance when $n_\alpha < k$. On the contrary, α may admit many expansions over β; as an example, if $\alpha = (2,3)$ and $\beta = (3,2,3)$ the corresponding expansions are given by

$$(1,1,3) \qquad (2,1,2) \qquad (2,2,1)$$

Also in this case, if $h = k$ then there is at most one expansion (actually in this case contraction and expansion of α over β coincide).

Now, let us define an algorithm to compute an expansion of a composition α over another composition β, where $\alpha = (a_i)_h$ and $\beta = (b_j)_k$ are given as input such that $h < k$ and $n_\alpha \leq n_\beta$ (the case $h = k$ is easy to deal with). The procedure first checks whether such an expansion exists. Note that here we have to compute a composition α' finer (and longer) than α. This means to group adjacent elements of β that correspond to each a_i.

Thus, the first step of the computation determines a sequence of sets $L_1, L_2, \ldots, L_h$, where each L_i contains the possible condidates for groups of adjacent b_t's that correspond to a_i. Formally, every L_i contains pairs $(j, \ell) \in \mathbb{N}^2$, where $1 \leq j \leq \ell \leq k$, such that

$$\ell - j + 1 \leq a_i \leq \sum_{t=j}^{\ell} b_t \tag{5}$$

$$(r, j-1) \in L_{i-1} \text{ for some } r \in \mathbb{N} \tag{6}$$

$$h - i \leq k - \ell \tag{7}$$

Condition (5) states that positive integers $a'_j, a'_{j+1}, \ldots, a'_\ell$ exist such that $a'_j \leq b_j, \ldots, a'_\ell \leq b_\ell$ and $\sum_{t=j}^{\ell} a'_t = a_i$. Condition (6) guarantees that the first $j-1$ elements of α' can be computed which correspond to the first $i-1$ elements of α. Finally, condition (7) assures that the remaining $k-\ell$ elements of α' (that are still to be computed) are enough to cover the remaining $h-i$ elements of α.

Therefore, we have:

$$
\begin{array}{lcl}
L_1 & = & \left\{(1,\ell)\in\mathbb{N}^2 \mid \ell \le a_1 \le \sum_{t=1}^{\ell} b_t\,,\; h-1 \le k-\ell\right\} \\
L_2 & = & \left\{(j,\ell)\in\mathbb{N}^2 \mid \ell-j+1 \le a_2 \le \sum_{t=j}^{\ell} b_t\,,\; h-2 \le k-\ell,\; \exists (s,j-1)\in L_1 \text{ for some } s\in\mathbb{N}\right\} \\
\dots & & \dots \\
L_i & = & \left\{(j,\ell)\in\mathbb{N}^2 \mid \ell-j+1 \le a_i \le \sum_{t=j}^{\ell} b_t\,,\; h-i \le k-\ell,\; \exists (s,j-1)\in L_{i-1} \text{ for some } s\in\mathbb{N}\right\} \\
\dots & & \dots \\
L_h & = & \left\{(j,\ell)\in\mathbb{N}^2 \mid \ell-j+1 \le a_h \le \sum_{t=j}^{\ell} b_t\,,\; \ell \le k,\; \exists (s,j-1)\in L_{h-1} \text{ for some } s\in\mathbb{N}\right\}
\end{array}
$$

Also here, it may occur that L_i is empty for some i: in this case all the subsequent L_j (with $j > i$) are empty and hence there is no expansion of α over β.

The following procedure computes all L_i's, where *Init* and *Next* play the same role as in the previous section.

```
begin
    Init := {1}
    for i = 1,2,...,h do
       begin
          Next := ∅
          L_i := ∅
          for j ∈ Init do
             begin
                ℓ := j
                x := 0
                while ℓ − j + 1 ≤ a_i ∧ h − i ≤ k − ℓ do
                   begin
                      x := x + b_ℓ
                      if a_i ≤ x then { add (j,ℓ) to L_i
                                      { if ℓ < k then Next := Next ∪ {ℓ + 1}
                      ℓ := ℓ + 1
                   end
             end
          if L_i = ∅ then { return no
                          { stop
          Init := Next
       end
end
```

Once the sequence $L_1, L_2, \dots, L_h$ is built, we go ahead as in the previous algorithm by choosing an element in each list and choosing the lists from the last one backwards to the first one. First, we look for an element (j,ℓ) in L_h such that $\ell = k$. If such a pair does not exist then there is no α' satisfying (3) and (4). Otherwise, such an expansion α' can be computed by the following procedure.

```
begin
    r := k
    for i = h, h-1, ..., 1 do
      begin
          find in L_i an element (j, ℓ) such that ℓ = r
          for t = j, ..., ℓ do a'_t := 1
          x := a_i - (ℓ - j + 1)
          t := j
                      ⎧ u := b_t - a'_t
                      ⎪ if x ≥ u      then a'_t := b_t
          while x > 0 do ⎨               else a'_t := a'_t + x
                      ⎪ x := x - u
                      ⎩ t := t + 1
          r := j - 1
      end
    return (a'_1, a'_2, ..., a'_k)
end
```

The first procedure works in time $O(hk^2)$ while the second one takes $O(k)$ steps once we maintain the sets L_i's as the S_i's in the previous section. By the same reason, both procedures can be implemented using a total space of the order $O(hk)$.

Computing expansions in quadratic time

However, also the previous algorithm can be improved and one can obtain an anologous procedure that works in $O(k^2)$ time. Let us now describe such an optimal version.

In this case our first task is the computation of all values B_{ij}, for indices $1 \le i \le j \le k$, such that

$$B_{ij} \;=\; \sum_{t=i}^{j} b_t$$

As before, this can be done in $O(k^2)$ time (for our convenience we assume $B_{ij} = 0$ for every $j < i$).

In a second phase, for every $i = 1, \ldots, h$, we compute a family of possible candidates for groups of adjacent b_t's corresponding to a_i. We look for pairs $(j, \ell) \in \mathbb{N}^2$, with $1 \le j \le \ell \le k$, that satisfy conditions (5), (6) and (7). However, rather than computing all possible pairs having these properties, more simply we fills up a table $L = \{L_{i\ell} \mid i = 1, \ldots, h,\ \ell = 1, \ldots, k\}$, where each entry $L_{i\ell}$ is the smallest j such that (j, ℓ) meets the required conditions. More precisely, L is defines as follows:

i) for every $\ell = 1, \ldots, k$,

$$L_{1\ell} = \begin{cases} 1 & \text{if } \ell \le a_1 \le B_{1\ell},\ h-1 \le k-\ell \\ 0 & \text{otherwise} \end{cases}$$

ii) For any $i = 2, \ldots, h$ and every $\ell = 2, \ldots, k$, setting

$$T = \{ j \in \mathbb{N} \mid 1 \le j \le \ell,\ \ell - j + 1 \le a_i \le B_{j\ell},\ L_{i-1\, j-1} \neq 0 \} ,$$

we have

$$L_{i\ell} \;=\; \begin{cases} \min\{j \in T\} & \text{if } T \neq \emptyset \text{ and } h - i \le k - \ell \\ 0 & \text{otherwise} \end{cases}$$

The computation of L is described by the following procedure. Again we proceed row by row. For a given row i, *Init* is the list of possible inital values of j that could be put in some entry $L_{i\ell}$. Here, the key observation is that for fixed i and j the constraints (5) and (7) allow us to look for a required ℓ by scanning backwards the interval $[j, k-h+i]$. This can be done efficiently by considering every ℓ such that $i \leq \ell \leq k-h+i$, at most once for all $j \in Init$.

```
begin
    Init := (1)
    for i = 1,2,...,h do
      begin
          New := Λ
          j := first(Init)
          ℓ₀ := j
          while  j ≠ null do
                begin
                    ℓ := min{k − h + i, a_i + j − 1}
                                                 ⎧ L_iℓ := j
                    while ℓ₀ ≤ ℓ ∧ a_i ≤ B_jℓ do ⎨ if ℓ < k then add ℓ + 1 to New
                                                 ⎩ ℓ := ℓ − 1
                    ℓ₀ := min{k − h + i, a_i + j − 1} + 1
                    j := next(j)
                end
          Init := New
      end
end
```

The analysis of the procedure can be carried on as in the previous section. Note that the inner loop is repeated at most $k-h+i$ times and it requires $O(1)$ time. As a consequence both the time and space complexity of the procedure are of the order $O(hk)$.

Once table L is filled in, one checks whether $L_{hk} > 0$; if this is not the case then there is no composition α' satisfying (3) and (4). Otherwise, such an expansion α' is computed by building a path backwards throughout the rows of L. The computation is described by the following procedure, which clearly works in $O(k)$ time.

```
begin
    ℓ := k
    for i = h, h − 1, ..., 1 do
      begin
          j := L_iℓ
          for t = j,...,ℓ do a'_t := 1
          x := a_i − (ℓ − j + 1)
          t := j
                         ⎧ u := b_t − a'_t
                         ⎪ if x ≥ u      then a'_t := b_t
          while x > 0 do ⎨               else a'_t := a'_t + x
                         ⎪ x := x − u
                         ⎩ t := t + 1
```

```
            ℓ = j − 1
        end
    return (a'_1, a'_2, ..., a'_k)
end
```

Note that, in this case, the computation of all B_{ij}'s, for $1 \leq i \leq j \leq k$, is the most expensive task of the algorithm. Hence the overal time and space complexity is $O(k^2)$.

4 Repeat-until languages

Given a finite alphabet Σ, let N be the set of all regular expressions over Σ such that:

i) every $a \in \Sigma$ belongs to N,

ii) if $\alpha, \beta \in N$ then $\alpha \cdot \beta \in N$ (often represented by $\alpha\beta$),

iii) if α is a symbol in Σ or an expression $\beta \cdot \gamma$, for some $\beta, \gamma \in N$, then $(\alpha)^+ \in N$.

We define a *repeat-until expression* as an expression $\alpha \in N$ containing just one occurrence of a for every $a \in \Sigma$. Thus, $\pi_\Sigma(\alpha)$ defines a linear order over Σ and, for every $a, b \in \Sigma$, we write $a < b$ if a occurs before b in $\pi_\Sigma(\alpha)$. We also denote by RUE the set of all repeat-until expressions over Σ.

For every $\alpha \in$ RUE, let $L(\alpha)$ be the language represented by α. Clearly, for every $x \in L(\alpha)$ and every $a, b \in \Sigma$, we have

$$a < b \text{ implies } \pi_{a,b}(x) \in a\{a,b\}^* b \tag{8}$$

Moreover, we define a *cycle* of α as a subexpression $(\beta)^+$ of α such that $\beta \in N$. The string $\pi_\Sigma(\beta)$ is the *body* of the cycle, $P(\pi_\Sigma(\beta))$ and $U(\pi_\Sigma(\beta))$ are its header and exit, respectively. For instance, $((ac)^+(bde)^+)^+$ is a cycle of $\alpha = h((ac)^+(bde)^+)^+ fg$, with header a and exit e.

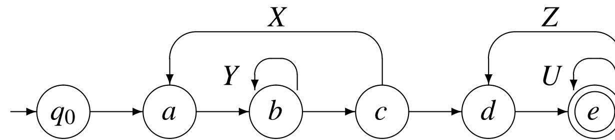
Note that in every $x \in L(\alpha)$ the body of any cycle appears at least once, possibly as a subword consisting of more factors. This justifies the definition of our expressions: any $\alpha \in$ RUE represents a program scheme of nested repeat-until cycles and every $x \in L(\alpha)$ represents an execution of the program.

Clearly, for any $\alpha \in$ RUE, $L(\alpha)$ is a local language [2]. A natural local automaton $\mathcal{A}(\alpha)$ recognizing $L(\alpha)$ can be obtained as follows. Given the string $\pi_\Sigma(\alpha) = a_1 a_2 \cdots a_m$, with $a_i \in \Sigma$ for each i, the set of states of $\mathcal{A}(\alpha)$ is $Q = \{q_0, a_1, \ldots, a_m\}$, where $q_0 \notin \Sigma$ is the initial state and a_m the unique final state. Also, the family of transitions E is given by the pairs

$$\begin{aligned} E &= \{(q_0, a_1)\} \cup \{(a_i, a_{i+1}) \mid i = 1, 2 \ldots, m-1\} \\ &\quad \cup \{(a_j, a_i) \mid a_i = P(\pi_\Sigma(\beta)), a_j = U(\pi_\Sigma(\beta)) \text{ for a cycle } (\beta)^+ \text{ of } \alpha\} \end{aligned}$$

Any transition $(a, b) \in E$ is labelled by the incoming state b. For any $q \in Q$ we also denote by $Suc(q)$ the family of its successors, i.e. the set $\{a \in \Sigma \mid (q, a) \in E\}$.

Example 1 *Consider the repeat-until expression $\alpha = (a(b)^+c)^+(d(e)^+)^+$. Then, the corresponding local automaton $\mathcal{A}(\alpha)$ is defined by the following diagram.*



where X, Y, Z, U represent the cycles $(a(b)^+c)^+$, $(b)^+$, $(d(e)^+)^+$ and $(e)^+$, respectively.

4.1 Hierarchical trees

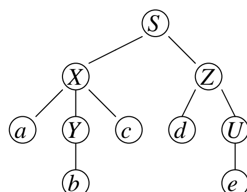
Here we describe a tree representation of expressions in RUE based on the cycles and the nesting relation. Given $\alpha \in$ RUE, let us represent the cycles of α by capitol letters and let $\mathcal{C}$ be the family of all of them together with a special symbol S, which will represent the root of the tree. For every $X, Y \in \mathcal{C}$, we define $X \trianglelefteq Y$ if X is nested into Y or $X = Y$. We also set $X \trianglelefteq S$ for every $X \in \mathcal{C}$. Moreover, we write $X \triangleleft Y$ if $X \trianglelefteq Y$ and $X \neq Y$.

Then we define the *hierarchical tree* of α as the ordered tree $T(\alpha)$ with root S, satisfying the following properties:

1. $\mathcal{C}$ is the set of internal nodes and $\pi_\Sigma(\alpha) = a_1 a_2 \cdots a_m$ is the ordered list of leaves;
2. For any $X, Y \in \mathcal{C}$, X is son of Y if $X \triangleleft Y$ and X is immediately nested in Y, i.e. there is no $Z \in \mathcal{C}$ such that $X \triangleleft Z \triangleleft Y$;
3. A leaf $a \in \Sigma$ is son of a node $X \in \mathcal{C}$ if X is the smallest cycle of α including a. If a is not included in any cycle then a is son of S;
4. since $T(\alpha)$ is an ordered tree, there is a linear order $<$ among the sons of any node $X \in \mathcal{C}$: given two sons u, v of X, $u < v$ if u (either as a cycle or as a letter in Σ) occurs before v in α.

Note that $X \trianglelefteq Y$ holds if X is descendant of Y in $T(\alpha)$.

Example 2 *The hierarchical tree of the repeat-until expression α defined in Example 1 is described by the following picture.*



For every $a \in \Sigma$, let $C(a)$ be the father of a in $T(\alpha)$: thus $C(a)$ either is the smallest cycle of α containing a or $C(a) = S$ if a is not included in any cycle. Analogously, for every $a, b \in \Sigma$, $a \neq b$, let $C(a,b)$ be the root of the smallest subtree of $T(\alpha)$ including both a and b. The following proposition states that all cycles are of the form $C(a)$ or $C(a,b)$ for some $a, b \in \Sigma$.

Proposition 1 *Let $\alpha \in RUE$ and let $X \in \mathcal{C}$ be a symbol different from S. Then, $X = C(a)$ for some $a \in \Sigma$ or $X = C(a,b)$ for some distinct $a, b \in \Sigma$.*

Proof. The property is proved by induction on the height of the node X in the hierarchical tree $T(\alpha)$. □

4.2 Syntactic trees

Now, given $\alpha \in$ RUE, let $\mathcal{C}$ and S be defined as in the previous section. Consider the grammar with regular right parts $G(\alpha)$ defined by the tuple $(\mathcal{C}, \Sigma, S, P)$, where $\mathcal{C}$ is the set of nonterminals, S is the initial symbol, Σ is the set of terminals and P is the family of productions given by

$$P = \{(X \to \gamma) \mid X \in \mathcal{C},\ \gamma \text{ is obtained from the list of sons of } X \text{ in } T(\alpha) \text{ by replacing each variable } Y \in \mathcal{C} \text{ by } Y^+\}$$

Example 3 *If* α *is defined as in Example 2 then*

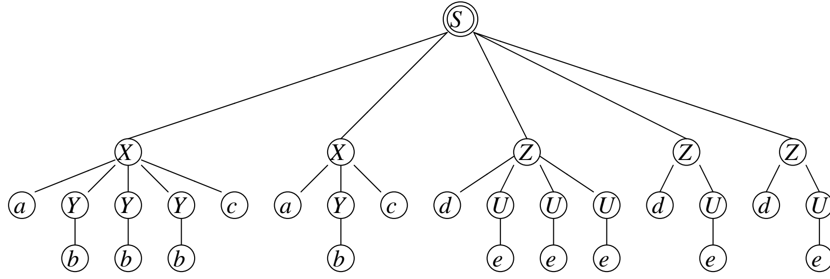
$$P = \{(S \to X^+Z^+), (X \to aY^+c), (Y \to b), (Z \to dU^+), (U \to e)\}$$

It is clear that $G(\alpha)$ generates $L(\alpha)$ in the usual way [9]. Thus, for any $x \in L(\alpha)$ we define the *syntactic tree* of x as the derivation tree of x in $G(\alpha)$. It corresponds to the *nested iterated tree* (NIT) in [12].

Example 4 *Let* α *be the repeat-until expression defined in Example 1 and let* x *be the string*

$$x = abbbcabcdeeedede$$

Then $x \in L(\alpha)$ *and its syntactic tree is given by the following picture:*



Proposition 2 *A word* $x \in \Sigma^*$ *belongs to* $L(\alpha)$ *if and only if there exists a syntactic tree* T *that generates* x.

Note that also the syntactic trees are ordered trees. They share several properties in common with the RUE tree $T(\alpha)$. First of all, they all have root S. Note that in $T(\alpha)$ there is just one node u for every $u \in \Sigma \cup \mathcal{C}$, while in a syntactic tree T there may be several nodes labelled by u: for the sake of brevity, they will be called *u-nodes* or *u-vertices*.

Moreover, if a node $u \in \Sigma \cup \mathcal{C}$ in $T(\alpha)$ is at a distance k from S then in T all u-nodes are at the distance k from the root. Since the u-vertices in T are ordered they can be identified by their occurrence number: if there are m nodes of label u the i-th u-node is univocally determined for any $i = 1, \ldots, m$.

Other properties of the syntactic tree T of a word $w \in L(\alpha)$ are the following:

1. For every $a \in \Sigma$, $|w|_a$ equals the number of nodes of T labelled by $C(a)$;

2. For every $a, b \in \Sigma$ with $a < b$, $|\pi_{a,b}(w)|_{ab}$ equals the number of nodes of T labelled by $C(a,b)$;

3. For every $a, b \in \Sigma$ with $a < b$, if $\alpha = (a_i)_h$ is the composition generated by $\pi_{a,b}(w)$ on a, then in T there are h nodes labelled by $C(a,b)$ and for any $i = 1, \ldots, h$ there are a_i nodes of label $C(a)$ that are descendants of the i-th node of label $C(a,b)$. Moreover, an analogous property holds for the composition generated by $\pi_{a,b}(w)$ on b.

Property 3 above actually shows that integer compositions can be used to represent an entire syntactic tree. To this end we introduce the notion of labelled composition.

Given a syntactic tree T, consider two cycles $A, B \in \mathcal{C}$ such that $B \trianglelefteq A$ and assume T has h nodes of label A and m nodes of label B. If $A \neq B$ define the labelled integer composition α_B^A by

$$\alpha_B^A = (a_1, a_2, \ldots, a_h)$$

where, for each $i = 1,\dots,h$, a_i is the number of B-nodes that are descendants of the i-th A-node in T. Clearly we have $m = n_{\alpha_B^A}$. On the contrary, if $A = B$ then set formally $\alpha_B^A = (1,1,\dots,1)$.

The symbols A and B are respectively the exponent and the base of α_B^A. It is clear that any syntactic tree is entirely described by the set of its labelled compositions. Actually a reduced set of such compositions would be sufficient to define a syntactic tree, since the other ones can be computed by using the operations of product or quotient, as shown by the following proposition, whose proof is consequence of the definitions.

Proposition 3 *Given a syntactic tree T, let A,B,C be cycles in $\mathcal{C}$ such that $C \trianglelefteq B \trianglelefteq A$. Then the following properties hold:*

$$
\begin{array}{lcl}
1)\ \alpha_C^B & \preceq & \alpha_C^A \\
2)\ \alpha_C^A & = & \alpha_B^A \cdot \alpha_C^B \quad \textit{and hence} \quad \alpha_B^A = \alpha_C^A / \alpha_C^B, \quad \alpha_C^B \in (\alpha_C^A \oslash \alpha_B^A)
\end{array}
$$

Further, if $A,B,C,D \in \mathcal{C}$ satisfy $D \trianglelefteq C \trianglelefteq B \trianglelefteq A$ then

$$
\begin{array}{ll}
3)\ \alpha_C^A & \textit{is a contraction of } \alpha_C^B \textit{ over } \alpha_D^A \textit{ and } \quad \alpha_D^C \in (\alpha_D^A \oslash \alpha_C^A) \\
4)\ \alpha_C^B & \textit{is an expansion of } \alpha_C^A \textit{ over } \alpha_D^B \textit{ and } \quad \alpha_D^C \in (\alpha_D^B \oslash \alpha_C^B)
\end{array}
$$

We also observe that the set of all labelled compositions of a given syntatic tree T contains the compositions of the form $\alpha_X^S = (k_X)$ for every $X \in \mathcal{C}$, where k_X is the number of X-nodes in T. Note in particular that, if X is father of Y in the hierarchical tree $T(\alpha)$ then α_Y^X is a matching of α_X^S and α_Y^S.

The previous properties can be used to construct a syntatic tree from a subset of its labelled compositions. A key property in such a construction is called coherence and concerns the inclusion relation $\preceq$ among labelled compositions having equal base. Let *Comp* be a set of labelled compositions. We say that *Comp* is referred to a hierachical tree $T(\alpha)$ if, for any $\alpha_B^A \in Comp$, the cycle B is descendant of A in $T(\alpha)$ and, for each $B \trianglelefteq A$, there is at most one composition α_B^A in *Comp*. We further say that *Comp* is *coherent* if for every pair of compositions $\alpha_C^A, \alpha_C^B \in Comp$, $B \trianglelefteq A$ implies $\alpha_C^B \preceq \alpha_C^A$. We know from Section 3 that coherence can be checked in linear time.

5 RUE trace language recognition

In this section we describe some general properties of trace languages defined by RUE expressions and show how they can be used to design algorithms for solving the corresponding recognition problem. We recall that, given an independence alphabet (Σ, I) and a RUE expression α on Σ, the membership problem for the trace language $[L(\alpha)] \subseteq M(\Sigma, I)$ consists of verifying, for an input $x \in \Sigma^+$, whether the set $[x] \cap L(\alpha)$ is empty.

Theorem 4 *Given a RUE expression α and an independence alphabet (Σ, I) with dependence relation D, for any $x \in \Sigma^+$ we have $[x] \cap L(\alpha) \neq$ if and only if the following conditions hold.*

a) *For every $a,b \in \Sigma$ such that $a < b$ and aDb, we have $\pi_{a,b}(x) \in a\{a,b\}^* b$.*

b) *There exists $w \in L(\alpha)$ having syntactic tree T such that:*

- **b1)** *For all $a \in \Sigma$, there are $|x|_a$ nodes labelled by $C(a)$ in T;*
- **b2)** *For every $a,b \in \Sigma$ such that aDb and $a < b$, $|\pi_{a,b}(x)|_{ab}$ equals the number of nodes of T labelled by $C(a,b)$;*

b3) *For any $a,b \in \Sigma$ such that aDb, let $(i_1,i_2,\dots,i_h)$ be the composition generated by $\pi_{a,b}(x)$ on a. Then, the first i_1 nodes labelled by $C(a)$ in T are descendants of the first $C(a,b)$-node, the subsequent i_2 nodes labelled by $C(a)$ are descendants of the second $C(a,b)$-node, and so on till the last i_h nodes labelled by $C(a)$, that are descendants of the last $C(a,b)$-node. Moreover, an analogous property holds for the composition generated by $\pi_{a,b}(x)$ on b.*

Proof. First recall that a word w belongs to $[x]$ if and only if $|x|_a = |w|_a$ for every $a \in \Sigma$ and $\pi_{a,b}(x) = \pi_{a,b}(w)$ for every pair of distinct symbols $a,b \in \Sigma$ such that aDb. Therefore, if there exists $w \in [x] \cap L(\alpha)$ then w satisfies condition (8) and properties 1, 2, 3 of Section 4.2. Since the projections of x and w on the pairs of (possible coincident) dependent symbols are equal, the same properties hold for x, proving both conditions a) and b).

On the other hand, if these two conditions are true then both x and w have the same projections on the pairs of (possible coincident) dependent symbols, and this proves that $w \in [x] \cap L(\alpha)$. □

A natural idea to solve the problem is to try to construct the syntactic tree T of a word $w \in [x] \cap L(\alpha)$. The computation may consist of two phases: first the nodes are determined, i.e. one calculates the number of X-nodes in T for every cycle X. Then, all edges are established by computing the labelled compositions α^A_B of T for every pair $A,B \in \mathcal{C}$ such that A is father of B in $T(\alpha)$.

5.1 Construction of the nodes

First of all, the root is the unique node labelled by S. Then, the leaves of T are determined by the occurrences of symbols of Σ in x: for every $a \in \Sigma$ one checks that $|x|_a \geq 1$ and adds $|x|_a$ leaves labelled by a in T.

As far as the internal nodes are concerned, it is clear that for every $X \in \mathcal{C}$, $X \neq S$, the number of X-nodes in T must satisfy conditions b1) and b2) of Theorem 4. This leads to consider the sets F_X and G_X defined by the following equations:

$$F_X = \{a \in \Sigma \mid X = C(a)\} \tag{9}$$

$$G_X = \{(a,b) \in \Sigma^2 \mid a < b, aDb, X = C(a,b)\} \tag{10}$$

If $F_X \neq \emptyset$ or $G_X \neq \emptyset$ we verify whether there exists $k_X \in \mathbb{N}$ such that $k_X = |x|_a$ for all $a \in F_X$, and $k_X = |\pi_{a,b}(x)|_{ab}$ for all $(a,b) \in G_X$. If both conditions are true, then any possible T contains k_X nodes labelled by X, otherwise such a tree does not exist.

However, if $F_X = G_X = \emptyset$ the previous computation cannot apply. In this case, to determine the number of X-nodes in T we can use the following proposition, stating that we are allowed to introduce as many X-node as the number of vertices in T labelled by the father of X in $T(\alpha)$.

Proposition 5 *Given $X \in \mathcal{C}$ such that $X \neq S$ and $F_X = G_X = \emptyset$, let Y be the father of X in $T(\alpha)$ and consider a word $w \in L(\alpha)$. Then, there exists $z \in L(\alpha) \cap [w]$ such that every Y-node in the syntactic tree of z has just one son labelled by X (and hence the number of X-nodes equals the number of Y-nodes).*

For the proof see Proposition 1 in [12].

Taking into account Propositions 1 and 5, we can summarize the previous discussion by the following program that constructs the set of internal nodes of T different from S.

```
begin
    B := ∅
```

for $X \in \mathcal{C} \backslash \{S\}$ do
 begin
  compute F_X and G_X
  if $F_X = G_X = \emptyset$ then add X to B
   else if $\exists k \in \mathbb{N}$ such that $k = |x|_a$ for every $a \in F_X$
    and $k = |\pi_{a,b}(x)|_{ab}$ for every $(a,b) \in G_X$
     then add k many X-nodes in T
     else reject x and stop
 end
for $X \in B$ (in order of distance from S) do
 begin
  $Y :=$ father of X in $T(\alpha)$
  $h :=$ number of Y-nodes in T
  add h many X-nodes in T
  for $i = 1, 2, \ldots, h$ do
   make the i-th X-node son of the i-th Y-node in T
 end
end

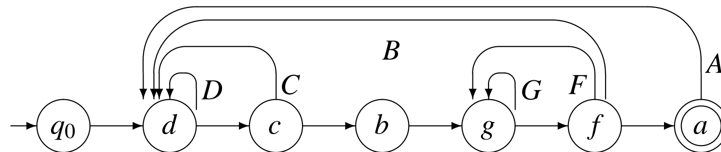
Note that the non-root internal nodes computed by the previous procedure are partitioned according to their labels. In particular, for every $X \in \mathcal{C} \backslash \{S\}$ there are k_X nodes in T labelled by X.

5.2 Construction of the edges: an example

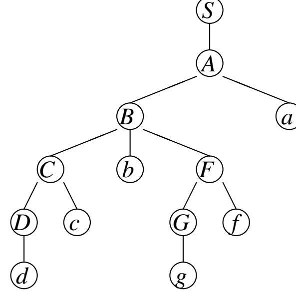
Once the nodes of T are computed, one can try to determine the father of each vertex (except for the root) by using the operations over labelled compositions introduced in Section 4.2. We are not able here to give a general procedure (as we did for the nodes) because the previous operations do not always yield a unique result and this ambiguity may affect the final result. We only present here an example that shows how to define an algorithm in specific cases when the structure of the hierarchical tree allows us to evoid ambiguities in the applications of the operations on compositions.

The idea is to use properties b1), b2) and b3) of Theorem 4 to define an initial set of compositions and then close such a set with respect to the operations of product, quotient, contraction, expansion and matching. In some cases the structure of the expression allows us to complete the tree even if some operations (contraction and expansion) admit multiple solutions.

Let $\alpha \in$ RUE be defined by the following diagram:



Here the set of cycle is defined by $\mathcal{C} = \{S, A, B, C, D, F, G\}$ and the correspoding hierarchial tree is given by



Moreover, assume that the dependency pairs are (a,d), (b,c), (f,g). Therefore the nodes of a possible syntactic tree are determined as in Section 5.1 by using the following relations:

$$A = C(a) = C(d,a)\ , \qquad B = C(b) = C(c,b)\ , \qquad C = C(c)$$
$$D = C(d)\ , \qquad F = C(f) = C(g,f)\ , \qquad G = C(g)$$

Thus, given an input $x \in \Sigma^*$, for each cycle $X \in \mathcal{C}$ different from S we can obtain the number k_X of X-nodes in the syntactic tree T of a possible $w \in [x] \cap L(\alpha)$. Note that in our case there is no $X \in \mathcal{C}$ such that $F_X = G_X = \emptyset$.

Then the contruction of the edges of T is described by the following computation that determines the set $Comp$ of compositions defining the tree.

1) $Comp := \emptyset$
2) for all $X \in \mathcal{C}$ add to $Comp$ the labelled composition $\alpha_X^S = (k_X)$
3) compute the labelled composition α_D^A generated by $\pi_{da}(x)$ over d
   compute the labelled composition α_C^B generated by $\pi_{bc}(x)$ over c
   compute the labelled composition α_G^F generated by $\pi_{gf}(x)$ over g
   add α_D^A, α_C^B and α_G^F to $Comp$
4) Check coherence of $Comp$
5) compute a contraction α_C^A of α_C^B over α_D^A and a corresponding matching α_D^C
   add both α_C^A and α_D^C to $Comp$ and check coherence
6) compute the quotient $\alpha_B^A = \alpha_C^A / \alpha_C^B$ and add it to $Comp$
7) compute a matching $\alpha_F^B = \alpha_F^S \oslash \alpha_B^S$ and add it to $Comp$

If any of the previous step cannot be completed then the procedure stops and rejects the input. Otherwise, at the end of the computation, the set $Comp$ contains all the labelled compositions α_Y^X for every pair father-son (X,Y) in $T(\alpha)$. Moreover, by construction $Comp$ is coherent and hence its closure with respect to the product yields the set of all labelled composition of the syntactic tree T of a word $w \in [x] \cap L(\alpha)$.

We conclude observing that the procedure works in time $O(n^2)$ where $n = |x|$, since this is the time required by step 5) while the other ones can be executed in linear time.

References

[1] A. Avellone, M. Goldwurm. Analysis of algorithms for the recongnition of rational and context-free trace languages. *RAIRO Theoretical Informatics and Applications* 32: 141-152, 1998.

[2] J. Berstel, J.-E. Pin. Local languages and the Berry-Sethi algorithm. *Theoret. Comput. Sci.* 155:439-446, 1996.

[3] A. Bertoni, M. Goldwurm, G. Mauri, N. Sabadini. Counting techniques for inclusion, equivalence and membership problems, in *The book of traces*, V. Diekert and G. Rozenberg Editors, World Scientific, 131-163, 1995.

[4] A. Bertoni, G. Mauri, N. Sabadini. Equivalence and memberchip problems for regular trace languages. Proc. 9th ICALP, LNCS 140: 61-71, Springer-Verlag, 1982.

[5] A. Bertoni, G. Mauri, N. Sabadini. Unambiguous regular trace languages. Proc. Coll. on Algebra, Combinatorics and Logic in Computer Science, Colloquia Mathematica Soc. J. Bolyai, 42: 113-123, North-Holland, 1985.

[6] A. Bertoni, G. Mauri, N. Sabadini. Membership problems for regular and context-free trace languages. *Information and Computation* 82 (2): 135-150, 1989.

[7] L. Breveglieri, S. Crespi Reghizzi, A. Savelli. Efficient word recognition of certain locally defined trace languages. Proc. 5th Int. Conf. on Words, Montreal (Canada), September 2005.

[8] P. Flajolet. Mathematical methods in the analysis of algorithms and data structures, in *Trends in Theoretical Computer Science*, E. Börger Editor, Computer Science Press, 225–304, 1988.

[9] W. LaLonde. Regular right part grammars and their parsers. *Communications of the ACM* 20 (10): 731–741, 1977.

[10] W. Rytter. Some properties of trace languages *Fund. Inform.* 7:117-127, 1984.

[11] J. Sakarovitch. On regular trace languages. *Theoret. Comput. Sci.* 52: 59-75, 1987.

[12] A. Savelli, Two contributions to automata theory on parallelization and data compression, Doctoral Thesis, Politecnico di Milano, Université de Marne-la-Vallée, June 2007.