UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament de Llenguatge i Sistemes Informàtics
Ph.D. Programme: Artificial Intelligence

SYMBOLIC AND CONNECTIONIST LEARNING TECHNIQUES FOR GRAMMATICAL INFERENCE

Autor: René Alquézar Mancho
Director: Alberto Sanfeliu Cortés

March 1997

Chapter 8

Augmented regular expressions: a formalism to describe and recognize a class of context-sensitive languages

In order to extend the potential of application of the syntactic approach to pattern recognition, the efficient use of models capable of describing context-sensitive structural relationships is needed, since most objects cannot be represented adequately by regular or context-free languages [BuSa:90, Tana:95]. Moreover, learning such models from examples is interesting to automate as much as possible the development of applications.

Context-sensitive grammars (CSGs) are not a good choice, since their parsing is computationally expensive and there is not any available algorithm to learn them automatically. *Augmented transition networks* (ATNs) [Woods:70] are powerful models that have been used in natural language processing, but which are very difficult to infer, as have been discussed in Section 3.3. *Pattern languages* [Angl:80b] provide a very limited mechanism to take into account some context influences, and some algorithms have been proposed to infer them from examples and queries (which have been reviewed in Section 3.4), but their expressive power is clearly insufficient since they are not even able to cover the regular languages.

On the other hand, it is known that *controlled context-free grammars* can generate some context-sensitive languages (CSLs) [Salo:73]. By using a recursive sequence of control sets on universal even-linear grammars, Takada showed that a hierarchy of language families that are properly contained in the class of CSLs can be learned using regular inference algorithms [Taka:94], as has been recalled in Section 3.2. Furthermore, an efficient parsing procedure can be devised for each language in any of these families, which is based on parsing successively by a set of universal even-linear grammars [Taka:94]. However, the gap in expressive power between each of these language families and the class of CSLs seems to be rather large, and it is not clear what types of context relations can be described by the controlled grammars.

We propose a new formalism to describe, recognize and learn a large subclass of CSLs: the so-called *augmented regular expressions* (or AREs). In this chapter, the augmented regular expressions are formally defined, their expressive power is studied, and an efficient method (low-polynomial in time) to recognize a string as belonging or not to the language represented by an unambiguous ARE is described. The material presented here is basically included in a paper that has been published recently in the *Pattern Recognition* journal [AlSa:97a], and also in a previous technical report [AlSa:95c] and a communication to the SSPR'96 workshop held in Leipzig (Germany) [SaAl:96]. The problem of learning AREs from examples and the methods proposed for this problem will be discussed in Chapter 9.

AREs are neither the *context-free expressions*, that are known to describe the family of CFLs, nor a type of regulated rewriting [Salo:73]. An ARE is a compact description based on a regular expression, in which the star symbols are replaced by natural-valued variables (called star variables) and these variables are related through a finite number of linear equations. Hence, AREs augment the expressive power of regular expressions (REs) by including a set of constraints, that involve the number of instances in a string of the operands of the star operations of an RE. In this way, REs are reduced to AREs with zero constraints among the star variables, and regular languages are obviously covered by the class of languages associated with AREs.

Although it is demonstrated that neither all CSLs nor all CFLs can be described by AREs, a large subclass of CSLs capable of describing planar shapes with symmetries and other context-dependent structures is representable, which is important for pattern recognition tasks. Likewise, it is proved that AREs cover all the pattern languages. The efficient method proposed for string recognition is split in two stages: parsing the string by the underlying RE and checking that the result of a successful RE parsing satisfies the constraints in the given ARE.

Prior to the formal definition of AREs, a preliminary section is included next that describes a method to select an RE equivalent to a given FSA, which is relevant in some cases for both parsing and learning AREs.

8.1 Obtaining regular expressions from FSAs

The formal definition of *regular expressions* (REs) has been given in Section 2.1.3, together with some of their properties and equivalence rules. It has also been remembered that REs and FSAs are alternative representations of the class of regular languages, and that some algorithms are known to find an FSA equivalent to a given RE and viceversa [Koha:78,HoUl:79].

The topic of obtaining REs from FSAs is discussed here because of two reasons related to AREs. First, in order to parse a string by an ARE, a previous parsing of that string by the underlying RE is needed to construct a certain data structure, and the parsing process can be made much more efficient if the RE has been obtained from a known equivalent DFA and some information about their relationship is available (see Section 8.4). Second, in order to learn an ARE from examples, the underlying RE must also be learned, and since the very most part of RGI methods do not infer an RE but an FSA, an FSA-to-RE mapping will be needed if any of those methods is used (see Chapter 9). Moreover, to facilitate the induction of AREs, the selection of an RE equivalent to a given FSA should be done taking into account some heuristic criteria. Hence, a specific algorithm is proposed as the mapping $\psi : FSA \rightarrow RE$ to be chosen in the context of learning AREs, whose outputs help to parse strings by the RE very efficiently.

On the other hand, the complexity of transforming an FSA into an RE is exponential in the worst case, and therefore, the use of RGI methods which directly return an RE is encouraged. It must be emphasized that, indeed, AREs can be inferred and parsed (still rather efficiently) without the need of an FSA-to-RE mapping for the underlying RE. In summary, the following practical guidelines are suggested concerning this point:

1) try to avoid the FSA-to-RE transformation in the method employed for inferring AREs;
2) if an FSA-to-RE mapping is performed anyhow, then use the algorithm that is proposed in this section;
3) if the transformation ends successfully in reasonable time with a reasonably long RE[1], then use the outputs of this algorithm for parsing strings by the resulting RE more efficiently.

The proposed mapping ψ is based on a classical method, the Arden's algorithm [Arden:60,Koha:78], but an inner modification and a final simplifyng step are included.

[1]In practice some timeout and/or some maximal length can be defined to stop the process.

The original Arden's algorithm is recalled in subsection 8.1.1, whereas the suggested changes are presented in 8.1.2 and 8.1.3.

8.1.1 The original Arden's algorithm

Let $A = (Q, \Sigma, \delta, q_0, F)$ be an FSA containing n states, and let $<$ be a total order defined among them, which can be arbitrary except that the first element must be the initial state q_0, i.e. $Q = (q_0, ..., q_{n-1})$. The order $<$ can also be applied to the final states, i.e. $F = (q_{f_1}, ..., q_{f_{|F|}})$, where $0 \leq f_1 < ... < f_{|F|} \leq n-1$.

Let α^l_{ij} denote the RE that describes the set of strings from Σ^* that take the automaton from state q_i to state q_j without passing through a state q_k with $k < l$. The RE denoted by α^n_{ij} will consist of the union of symbols from Σ that correspond to direct transitions from q_i to q_j (or $\emptyset$, if there is no direct transition), whereas the RE denoted by α^0_{ij} will describe the whole set of strings that lead from q_i to q_j. Let R_j be a synonim for α^0_{0j}, which denotes the RE that describes the set of strings that take the automaton from the initial state q_0 to a state q_j.

Then, it is clear that a valid RE R equivalent to A is given by

$$R = \sum_{q_f \in F} \alpha^0_{0f} = R_{f_1} + ... + R_{f_{|F|}} \tag{8.1}$$

Hence, a procedure that determines all the R_j, for $0 \leq j < n$, can be used to yield R. The Arden's method (Algorithm 8.1) is such a procedure, which is based on solving the following system of symbolic equations:

$$\begin{array}{lcllllllll}
(e^n_0): & R_0 & = & R_0\alpha^n_{00} & + & R_1\alpha^n_{10} & + \; ... \; + & R_{n-1}\alpha^n_{(n-1)0} & + & \lambda \\
(e^n_1): & R_1 & = & R_0\alpha^n_{01} & + & R_1\alpha^n_{11} & + \; ... \; + & R_{n-1}\alpha^n_{(n-1)1} & & \\
... & ... & & ... & & ... & ... & ... & & \\
(e^n_{n-1}): & R_{n-1} & = & R_0\alpha^n_{0(n-1)} & + & R_1\alpha^n_{1(n-1)} & + \; ... \; + & R_{n-1}\alpha^n_{(n-1)(n-1)} & &
\end{array}$$

where e^n_j are labels to identify each equation. This system can be solved in n steps. At each step from $j = n-1$ down to $j = 0$, equation (e^{j+1}_j) is processed using the rule given by Theorem 2.2, $\lambda \notin L(P) \Rightarrow (R_j = Q + R_jP \Leftrightarrow R_j = QP^*)$, and then R_j is substituted into the rest of equations (e^{j+1}_i), $i \neq j$. Whenever Theorem 2.2 (Arden's lemma) is not applicable, the right hand side of the equation (e^{j+1}_j) can be directly used to replace R_j.

ALGORITHM 8.1: *Arden's method for finding an RE equivalent to an FSA.*

Inputs:

A is a given FSA $(Q, \Sigma, \delta, q_0, F)$ with an ordered set of states Q;

n is the number of states of A;

Outputs:

α^l_{ij} $(1 \leq l \leq n,\ 0 \leq j < n,\ 0 \leq i < l)$, are REs associated with different sets of paths of A, where α^l_{ij} describes the language of strings that take A from state q_i to state q_j without passing through a state q_k with $k < l$;

R_j $(0 \leq j < n)$, are the REs that describe the languages corresponding to each one of the states of A;

R is the returned RE equivalent to A;

```
begin_algorithm
{ build the initial system of equations }
for $j := 0$ to $n-1$ do
    for $i := 0$ to $n-1$ do
        Find $\{a_k \mid a_k \in \Sigma$ and $q_j \in \delta(q_i, a_k)\}$
        $\alpha^n_{ij} := \sum a_k$
    end_for
    if $j = 0$ then
        Write eq.($e^n_j$) as "$R_0 = R_0\alpha^n_{00} + ... + R_{n-1}\alpha^n_{(n-1)0} + \lambda$"
    else
        Write eq.($e^n_j$) as "$R_j = R_0\alpha^n_{0j} + ... + R_{n-1}\alpha^n_{(n-1)j}$"
    end_if
end_for

{ solve the equations }
for $l := n-1$ down to $1$ do
    { solve eq.($e^{l+1}_l$) : $R_l = R_0\alpha^{l+1}_{0l} + ... + R_{l-1}\alpha^{l+1}_{(l-1)l} + R_l\alpha^{l+1}_{ll}$ }
    if $\alpha^{l+1}_{ll} \neq \emptyset$ then
        { Apply Arden's lemma to eq.($e^{l+1}_l$), $R_l = (R_0\alpha^{l+1}_{0l} + ... + R_{l-1}\alpha^{l+1}_{(l-1)l})\ \alpha^l_{ll}$ }
        $\alpha^l_{ll}$ := form_star_type_RE($\alpha^{l+1}_{ll}$)   { i.e. $\alpha^l_{ll} := \alpha^{l+1*}_{ll}$ }
        for $i := 0$ to $l-1$ do
            if $\alpha^{l+1}_{il} \neq \emptyset$ then $\alpha^l_{il} := \alpha^{l+1}_{il}\alpha^l_{ll}$
            else                        $\alpha^l_{il} := \emptyset$
            end_if
        end_for
    else
        for $i := 0$ to $l-1$ do
            $\alpha^l_{il} := \alpha^{l+1}_{il}$
        end_for
    end_if
    Write eq.($e^l_l$) as "$R_l = R_0\alpha^l_{0l} + ... + R_{l-1}\alpha^l_{(l-1)l}$"
```

```
        { substitute the right hand side of eq.($e_l^l$) for $R_l$ into the rest of equations }
        for $j := 0$ to $n-1$ do
            if $j \neq l$ then
                if $\alpha_{lj}^{l+1} \neq \emptyset$ then
                    for $i := 0$ to $l-1$ do
                        if $\alpha_{il}^{l} \neq \emptyset$ then
                            if $\alpha_{ij}^{l+1} \neq \emptyset$ then
                                $\alpha_{ij}^{l} := (\alpha_{ij}^{l+1} + \alpha_{il}^{l}\alpha_{lj}^{l+1})$
                            else
                                $\alpha_{ij}^{l} := \alpha_{il}^{l}\alpha_{lj}^{l+1}$
                            end_if
                        else
                            $\alpha_{ij}^{l} := \alpha_{ij}^{l+1}$
                        end_if
                    end_for
                else
                    for $i := 0$ to $l-1$ do
                        $\alpha_{ij}^{l} := \alpha_{ij}^{l+1}$
                    end_for
                end_if
                if $j = 0$ then
                    Write eq.($e_j^l$) as "$R_0 = R_0\alpha_{00}^{l} + ... + R_{l-1}\alpha_{(l-1)0}^{l} + \lambda$"
                else
                    Write eq.($e_j^l$) as "$R_j = R_0\alpha_{0j}^{l} + ... + R_{l-1}\alpha_{(l-1)j}^{l}$"
                end_if
            end_if
        end_for
    end_for
    { solve the equation ($e_0^1$) : $R_0 = R_0\alpha_{00}^1 + \lambda$ }
    if $\alpha_{00}^1 \neq \emptyset$ then $R_0 := \alpha_{00}^{1*}$
    else                               $R_0 := \lambda$
    end_if
    { substitute $R_0$ in the rest of equations }
    for $j := 1$ to $n-1$ do
        if $\alpha_{0j}^1 \neq \emptyset$ then $R_j := R_0\alpha_{0j}^1$
        else                               $R_j := \emptyset$
        end_if
    end_for
    { build $R$ from the REs $R_f$ corresponding to the final states }
    $R := \sum_{q_f \in F} R_f$     { apply Eq.(8.1) }
    end_algorithm
```

In the preceding algorithm, the function *form_star_type_RE*, that is called in the application of Arden's lemma, is simply defined as

```
function form_star_type_RE ($\alpha_{ll}^{l+1}$) returns RE
begin
return $\alpha_{ll}^{l+1*}$
end_function
```

Algorithm 8.1 calculates all the REs R_j $(0 \leq j < n)$, and therefore it is able to obtain the equivalent R as given by Eq.(8.1). In addition, it also computes the REs α_{ij}^{l} $(1 \leq l \leq n,\ 0 \leq j < n,\ 0 \leq i < l)$, although some of these REs may be empty. A subset of $2n^2$ of the REs α_{ij}^{l} are used by an algorithm described in Section 8.4 to parse a string by an RE efficiently with the help of a source DFA. More precisely, the required REs are those denoted by α_{ij}^{j+1}, α_{ji}^{j+1}, α_{ij}^{n}, α_{ji}^{n} for $1 \leq j \leq (n-1)$, $0 \leq i < j$, and α_{jj}^{j+1}, α_{jj}^{n} for $0 \leq j \leq (n-1)$. Consequently, Algorithm 8.1 can save each of these REs once it is computed, needing to store at each step only the REs α_{ij}^{l} for the current value of l, i.e. a number of n^2 REs.

However, it must be noticed that the time complexity of Arden's algorithm is exponential $O(2^n)$ in the number of states of the given FSA in the worst case, due to the fact that the length of the equivalent RE might be exponential in n. This occurs for example when the FSA is fully-connected (i.e. its state transition diagram is a clique). Nevertheless, in many cases, when the given FSA presents some limitations on the connectivity and degree of circuit embedment in its state transition graph, a run-time polynomial in n can be achieved in practice (e.g. for FSA equivalent to REs of the form $a_1^*a_1...a_i^*a_i...a_n^*a_n$ $(a_i \in \Sigma)$, a run-time cubic in n is obtained). Indeed, a best-case complexity of $\Omega(n^3)$ can be shown by realizing that a number cubic in n of REs α_{ij}^{l} are computed.

8.1.2 A modification of Arden's algorithm to discriminate loops from other circuits in the resulting RE

The application of Arden's algorithm to a DFA gives rise to an *unambiguous* RE, where an RE R is said to be unambiguous if for all the strings $s \in L(R)$ a unique parsing of s by R can be made. This is an important feature that permits the use of efficient parsing methods for REs derived from DFAs, as will be explained in Section 8.4.

Note, however, that we could further transform the RE R returned by Arden's method into some other REs, which would also be equivalent to the given FSA, by applying some RE equivalence rules like those given by Eqs.(2.1) to (2.15) in Section 2.1.3. This makes sense if there is some kind of preference within equivalent REs that can be translated into a well-defined sequence of equivalence rules to apply.

In order to be able to induce and represent the greatest number of constraints using the ARE formalism, the underlying RE should be selected among the REs in its equivalence class according to the following two (somewhat opposite) heuristics:

1. Maximize the number of star symbols.
2. Preserve unambiguity.

The aim of the first heuristic is to increase the potential for inferring context relations that involve the number of consecutive matches of the star operands in parsing strings by the RE. The aim of the second one is to ease both the RE parsing and constraint induction processes.

It can be observed that applying any of the rules $P^* = P^*P^*$, $P^* = (P^*)^*$, $(P+Q)^* = (P^*+Q^*)^*$, $(P+Q)^* = (P^*Q^*)^*$, to an RE leads to an equivalent RE with a larger number of stars. However, these rules introduce a great deal of ambiguity in the resulting RE. For example, let $P^* = a^*$ and take a^5 as input string, it follows that a^0a^5, a^1a^4, a^2a^3, ... are possible ways of parsing by a^*a^*, and $a^1a^1a^1a^1a^1$, $a^2a^0a^1a^2$, $a^1a^3a^1$, ... are possible ways of parsing by $(a^*)^*$.

On the other hand, the equivalence rule $(P+Q)^* = (P^*Q)^*P^*$, given by Eq.(2.15), can be used to increase the number of stars and does not only preserve but even enforces unambiguity, since an RE containing a union operation is transformed into an RE containing just concatenation and star operations. For instance, let $(P+Q)^* = (a+b)^*$ and take $a^3ba^3ba^3$ as input string, it results that $(a+b)^{11}$ is the unique parsing by $(a+b)^*$ (in which the repetitive pattern a^3b is not reflected) whereas $(a^3b)^2a^3$ is the unique parsing by $(a^*b)^*a^*$.

All the star symbols in the output RE of Algorithm 8.1 are originated by instantiating the rule of Theorem 2.2 in solving the equations (e_l^{l+1}), $0 \leq l < n$, this is

$$R_l = (R_0\alpha_{0l}^{l+1} + \ldots + R_{l-1}\alpha_{(l-1)l}^{l+1}) + R_l\alpha_{ll}^{l+1} \Rightarrow R_l = (R_0\alpha_{0l}^{l+1} + \ldots + R_{l-1}\alpha_{(l-1)l}^{l+1})\,\alpha_{ll}^{l+1*}$$

Therefore, if the RE α_{ll}^{l+1} is of the form $(P+Q)$, we may apply Eq.(2.15) to the subexpression α_{ll}^{l+1*} to yield a "better" result, in terms of the above heuristics. In the general case, there can be many ways of decomposing α_{ll}^{l+1} into the P and Q union operands. A meaningful decomposition is given by $P = \alpha_{ll}^{n}$ and $Q = \alpha_{ll}^{l+1} - \alpha_{ll}^{n}$, where

P denotes the direct transitions from q_l to itself (the *loops* of q_l) and Q denotes the *circuits* starting and ending in q_l that only can traverse states q_k with $k > l$.

Hence, the proposed modification consists of replacing the previous definition of the function *form_star_type_RE*, which is called in Algorithm 8.1, by the following one:

```
function form_star_type_RE ($\alpha_{ll}^{l+1}$) returns RE
begin
if $\alpha_{ll}^{l+1} = \alpha_{ll}^{n} \vee \alpha_{ll}^{n} = \emptyset$ then
      return $\alpha_{ll}^{l+1*}$      { do not decompose $\alpha_{ll}^{l+1}$ }
else     { $\alpha_{ll}^{l+1} = (\alpha_{ll}^{n} + Q)$ }
      $Q := \alpha_{ll}^{l+1} - \alpha_{ll}^{n}$     { actually remove the term(s) of $\alpha_{ll}^{n}$ from the
                               union-type RE $\alpha_{ll}^{l+1}$ }
      return $(\alpha_{ll}^{n*}Q)^*\alpha_{ll}^{n*}$     { apply Eq.(2.15), where $P = \alpha_{ll}^{n}$ }
end_if
end_function
```

In this way, loops are discriminated from the rest of circuits of the given FSA in the resulting equivalent RE. This can be interesting for pattern recognition tasks, where the loops of an FSA model usually account for (indefinite) length or duration of a basic primitive, a meaningful structure in the pattern, specially if tools are provided to relate the lengths, or durations, of the different parts. The usefulness of the proposed modification for inferring AREs will be illustrated with an example that is presented in Section 9.1.

8.1.3 Simplifying the regular expression obtained by Arden's algorithm

For both the original Arden's algorithm and the modified version that has been described, the RE R that is obtained at the end by applying Eq.(8.1) can be simplified a lot by determining the common prefixes in the terms of the union operation and using the rule $PQ + PS = P(Q + S)$ repeatedly. Next, it is explained how to build the simplified equivalent RE, that results from the extraction of common prefixes, directly from some of the REs α_{ij}^{l} computed by Algorithm 8.1.

First, let us find a simplified RE for the union of the regular languages R_j for all the states $q_j \in Q$ of the given FSA A. By replacing recursively the R_i variables in equations (e_j^j) by the expressions given in equations (e_i^i), for $0 < j < n$ and $0 < i < j$, the following equivalence is obtained:

$$\sum_{q_j \in Q} R_j = R_0 + \dots + R_{n-1} = R_0(\lambda + R_{01}P^Q_{1(n-1)} + \dots + R_{0(n-1)}P^Q_{(n-1)(n-1)}) = R_0P^Q_{0(n-1)} \tag{8.2}$$

where $R_{ij} = \alpha^j_{ij}$ and P^Q_{ik} denotes an RE that represents the set of strings that lead from state q_i to any state $q_j \in Q$ with $i \le j \le k$ without passing through a state q_l with $l \le i$. The RE P^Q_{ik} can be defined recursively as

$$P^Q_{ik} = \lambda + \sum_{j=i+1,k}^{R_{ij}\neq\emptyset} R_{ij}\ P^Q_{jk} \tag{8.3}$$

Now, we define a relation $q_i \sim q_j$ in the following way:

$$q_i \sim q_j \Leftrightarrow (j < i) \wedge R_{ji} \neq \emptyset$$

Let $Q' \subseteq Q$ be any subset of states, and let $C(Q')$ be the transitive closure of Q' with respect to the relation $\sim$. It turns out that

$$\forall i,\ 0 \le i < n,\ \forall k,\ i \le k < n: \quad P^{Q'}_{ik} = \emptyset \Leftrightarrow q_i \notin C(Q'). \tag{8.4}$$

Hence, $P^{Q'}_{ik}$ can be computed recursively using the closure set $C(Q')$ for filtering null terms as follows

$$P^{Q'}_{ik} = \begin{cases} \lambda + \displaystyle\sum_{j=i+1,k}^{R_{ij}\neq\emptyset\ \wedge\ q_j\in C(Q')} R_{ij}\ P^{Q'}_{jk} & \text{if } q_i \in Q' \\ \displaystyle\sum_{j=i+1,k}^{R_{ij}\neq\emptyset\ \wedge\ q_j\in C(Q')} R_{ij}\ P^{Q'}_{jk} & \text{otherwise} \end{cases} \tag{8.5}$$

Then the union of the regular languages associated with the states of Q' is equivalent to a simplified RE with the extracted common prefixes:

$$\sum_{q_j \in Q'} R_j = R_0 P^{Q'}_{0(n-1)} \tag{8.6}$$

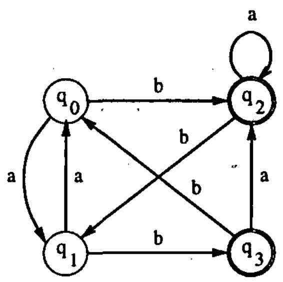
Furthermore, if $Q' \neq \emptyset$ then $P^{Q'}_{0(n-1)} = P^{Q'}_{0z}$, where q_z is the state of Q' with the largest index.

Finally, the RE $R = \psi(A)$ that is selected among the REs equivalent to the given automaton A can be expressed as

$$R = R_0 P^F_{0f_{|F|}} \tag{8.7}$$

where the set of final states F plays the role of the subset Q', and therefore, R is equivalent to the RE $\sum_{q_f \in F} R_f$.

We refer to the string that is obtained by substituting in R the REs corresponding to R_0 and each of the R_{ij} by some special symbols "R0" and "Ri_j" as the *skeleton* of R. This skeleton, $skel(R)$, can be constructed in parallel with the RE R after running Arden's algorithm, using Eqs.(8.7) and (8.5).



a)

$$\begin{aligned} R_0 &= ((a+ba^*b)(baa^*b)^*(a+bb))^* \\ R_1 &= ((a+ba^*b)(baa^*b)^*(a+bb))^*\ (a+ba^*b)(baa^*b)^* \\ R_2 &= ((a+ba^*b)(baa^*b)^*(a+bb))^*\ (ba^*+(a+ba^*b)(baa^*b)^*baa^*) \\ R_3 &= ((a+ba^*b)(baa^*b)^*(a+bb))^*\ (a+ba^*b)(baa^*b)^*b \end{aligned}$$

b)

$$\begin{aligned} R_2+R_3 = \ & ((a+ba^*b)(baa^*b)^*(a+bb))^*\ (ba^*+(a+ba^*b)(baa^*b)^*baa^*)\ + \\ & ((a+ba^*b)(baa^*b)^*(a+bb))^*\ (a+ba^*b)(baa^*b)^*b \end{aligned}$$

c)

$$R = R_0\ P_{03}^{\{q_2,q_3\}} = ((a+ba^*b)(baa^*b)^*(a+bb))^*\ ((a+ba^*b)(baa^*b)^*\ (baa^*\ +\ b)\ +\ ba^*)$$

d)

$$skel(R) = R0\ (R0\_1\ (R1\_2\ +\ R1\_3)\ +\ R0\_2)$$

e)

Fig. 8.1 a) *An FSA $A = (Q, \Sigma, \delta, q_0, F)$ with 4 states, where $F = \{q_{f_1}, q_{f_2}\} = \{q_2, q_3\}$*
b) *The regular languages associated with the states of A given by Algorithm 8.1*
c) *The straightforward RE for A: $\sum_{q_f \in F} R_f$*
d) *The simplified RE for A: $R = \psi(A) = R_0\ P^F_{0f_{|F|}}$*
e) *The skeleton of R*

Consider, for example, the 4-state DFA that is displayed in Fig. 8.1a). The two equivalent REs, given by Eqs.(8.1) and (8.7), that are obtained for this DFA, are shown in Fig. 8.1c) and d), respectively. The skeleton of the simplified RE is displayed in Fig. 8.1e).

8.2 Augmented regular expressions (AREs)

In order to define the *augmented regular expressions* (AREs) some preliminary concepts are needed, which are introduced in the following subsections.

8.2.1 The star variables and the star tree of a regular expression

Definition 8.1. Let R be a given RE and let us say that R includes ns star symbols ($ns \geq 0$). The set of *star variables* associated with R is an ordered set of natural-valued variables $V = \{v_1, ..., v_{ns}\}$, which are associated one-to-one with the star symbols that appear in R in a left-to-right scan.

Let $pos(V, i)$ be a function that returns the position in R of the star symbol associated with the star variable v_i. Actually, this position can be represented as an attribute of each star variable. Moreover, if p is the position in R of a star symbol, then the inverse function $pos^{-1}(V, p)$ returns the index i of the corresponding star variable v_i. The function pos is used to order the set V: $v_i \prec v_j \Leftrightarrow pos(V, i) < pos(V, j)$.

Definition 8.2. For $v_i, v_j \in V$, we say that v_i *contains* v_j iff the operand of the star associated with v_i in R includes the star symbol corresponding to v_j; and we say v_i *directly-contains* v_j iff v_i contains v_j and there is no $v_k \in V$ such that v_i contains v_k and v_k contains v_j.

Definition 8.3. Given an RE R, its associated *star tree* $\mathcal{T} = (N, E, r)$ is a general tree in which the root node r is a special symbol, the set of nodes is $N = V \cup \{r\}$, and the set of edges E is defined by the containment relationships of the star variables:

i) an edge (r, v_i) is created for each $v_i \in V$ that is not *directly-contained* by other star variable (where v_i is said to be located in the first level of the tree and to be a *son* of the root r, which is at level 0);

ii) for all $v_i, v_j \in V$, if v_i *directly-contains* v_j then an edge (v_i, v_j) is created (where v_j is said to be a *son* of v_i).

Furthermore, let us assign an integer identifier to each node of the star tree $\mathcal{T}$: let the identifier of the root r be 0, and let the identifier of any other node be the index i of the star variable v_i corresponding to that node ($1 \leq i \leq ns$). Algorithm 8.2 describes how to build the *star tree* $\mathcal{T}$ associated with a given RE R; its time complexity is $O(|R| \cdot h(R))$, where $h(R)$ is the depth of non-removable parentheses in R.

ALGORITHM 8.2: *Builds the star tree associated with a given RE.*

```
Inputs:
    R   is a given regular expression over an alphabet $\Sigma$;
Outputs:
    V   is the set of star variables associated with R, that is ordered by
        the position of the corresponding stars in R and contains this
        position information, which is required for the functions pos and $pos^{-1}$;
    $\mathcal{T}$   is the star tree associated with R.
begin_algorithm
V := obtain_star_variables_of_RE (R)  { build a representation for V that includes the star
    positions by scanning R from left to right }
r := create_startree_node(0)  { create a node with 0 as identifier and no son }
$\mathcal{T}$ := tree_whose_root_is(r)  { initialise the star tree $\mathcal{T}$ with only the root node }
concatenation_analysis (R, 0, V, $\mathcal{T}$, r)  { analyse R as a concatenation of REs, linking its
    outermost star variables as sons of r and their corresponding trees as subtrees of $\mathcal{T}$ }
end_algorithm
```

```
procedure concatenation_analysis (P, p0, V, $\mathcal{T}$, father)
input arguments:
    P is an RE over $\Sigma$ included in R;
    p0 is the position of P in the whole RE R;
    V is the set of star variables associated with R;
input/output arguments:
    $\mathcal{T}$ is the star tree associated with R that is being built;
    father is the node of $\mathcal{T}$ that is the root of the current subtree being built;
begin
U := find_factors(P)  { U is a list of triples ($< u_1, p_1, l_1 > ... < u_m, p_m, l_m >$) where the
    $u_i$'s are REs such that $P = u_1 ... u_m$, $m \geq 1$, $p_i$ is the position of the RE $u_i$ in P, and
    $l_i$ is the length of $u_i$ }
reset_list(U);
```

```
while not end_of_list($U$) do
    $< u, p, l >$:= get_current_element($U$);
    if is_a_terminal_symbol($u$) { $u = a$ for some $a \in \Sigma$ } then { Do nothing }
    elseif is_a_simple_startype_RE($u$) { $u = a^*$ for some $a \in \Sigma$ } then
        $pp := p0 + p + 1$; { is the position of star $*$ in $R$ }
        $j := pos^{-1}(V, pp)$; { is the index of the associated star variable $v_j$ }
        $node$ := create_startree_node($j$); { with $j$ as identifier and no son }
        create_edge_in_startree($\mathcal{T}, father, node$); { from father to node }
    elseif is_a_uniontype_RE($u$) { $u = (P')$ where $(P')$ is an RE over $\Sigma$ } then
        union_analysis ($u, p0 + p, V, \mathcal{T}, father$); { analyse $u$ as a union of REs, linking
                its outermost star variables as sons of $father$ and creating their subtrees}
    elseif is_a_parenthesised_startype_RE($u$) { $u = (P')^*$ } then
        $pp := p0 + p + l - 1$; { is the position of star $*$ in $R$ }
        $j := pos^{-1}(V, pp)$; { is the index of the associated star variable $v_j$ }
        $node$ := create_startree_node($j$); { with $j$ as identifier and no son }
        create_edge_in_startree($\mathcal{T}, father, node$); { from father to node }
        $u$ := remove_star_from($u$); { $u = (P')$ }
        union_analysis ($u, p0 + p, V, \mathcal{T}, node$); { analyse $(P')$ as a union of REs, linking
                its outermost star variables as sons of $node$ and creating their subtrees }
    end_if
    move_to_next_element($U$);
end_while
end_procedure
```

```
procedure union_analysis ($P, p0, V, \mathcal{T}, father$)
input arguments:
    $P$ is an RE over $\Sigma$ of the form $(P')$;
    $p0$ is the position of $P$ in the whole RE $R$;
    $V$ is the set of star variables associated with $R$;
input/output arguments:
    $\mathcal{T}$ is the star tree associated with $R$ that is being built;
    $father$ is the node of the star tree $\mathcal{T}$ that is the root of the subtree being built;
begin
$T$ := find_terms($P$); { $T$ is a list of triples $(< t_1, p_1, l_1 > \ldots < t_n, p_n, l_n >)$ where the $t_i$'s
    are REs such that $P = (t_1 + \ldots + t_n)$, $n \geq 1$, $p_i$ is the position of the RE $t_i$ in $P$, and
    $l_i$ is the length of $t_i$ }
reset_list($T$);
while not end_of_list($T$) do
    $< t, p, l >$:= get_current_element($T$);
    concatenation_analysis ($t, p0 + p, V, \mathcal{T}, father$); { analyse $t$ as a concatenation of
        REs, linking its outermost star variables as sons of $father$ and creating their
        subtrees}
    move_to_next_element($T$);
end_while
end_procedure
```

Definition 3.4. We say that a star variable $v \in V$ is *instantiated*, during the parsing of a string s by the RE R from which V has been defined, each time the operand of the corresponding star (an RE) is matched some number of consecutive times (possibly zero) against a substring of s. The number of consecutives matches (*cycles*) of the star operand in an instance of v will be the *value* of v for that instance.

Hence, star variables can only take natural numbers as values. However, we will see that, for computational purposes, it is useful to assign a special value, say -1, to a star variable v, whenever v is not instantiated during a cycle of an instance of its father in $\mathcal{T}$. This can occur when union-type REs included in R are matched against substrings of s. For example, if a star operand in R associated with a star variable v_f consists of a union of two or more REs (called *terms*), it can be traced during parsing which term is used for each match of the operand, and therefore, for each cycle of an instance of v_f, only the star variables that are located in the matched term can be actually instantiated, whereas the special value -1 will be given to the rest of star variables that are *directly-contained* by v_f (sons of v_f in $\mathcal{T}$). In this way, all the star variables that are brothers in the star tree $\mathcal{T}$ will have the same structure of *instances* for a given string, and we will distinguish between the *actual instances* (those with a natural value) and the *dummy instances* (those with value -1).

In the next subsection, a data structure is presented which is designed to store the information of the instances of the star variables that occur in parsing a string by a regular expression.

8.2.2 The star instances data structure

Given an RE R over an alphabet Σ and a string $s \in \Sigma^*$, it may be desired to parse s with respect to R. A parsing algorithm must return *yes* or *not* depending on whether $s \in L(R)$ or not, and in the first case, it must also return a kind of "instantiation" of R that just describes s, i.e. something similar to a derivation tree in the case of parsing a string by a grammar. The *star instances* data structure, which is defined next, can be regarded as a *partial* representation of such an "instantiation", since the matched sub-expressions theirselves are not recorded[2].

Let $V = \{v_1, ..., v_{ns}\}$ be the ordered set of star variables associated with an *unambiguous* RE R. Given a certain string s belonging to the language $L(R)$, a data

[2]Indeed, some information of the RE parsing may be lost if some of the matched sub-expressions belong to a union-type RE and do not contain any star symbol, but this information is not relevant for parsing by an ARE (see Section 8.4).

structure $SI_s(V) = \{SI_s(v_1), ... SI_s(v_{ns})\}$, called the set of *star instances* of the star variables in V for s, can be built during the process of parsing s by R. If the RE R were *ambiguous*, then a different set of star instances $SI_s^p(V)$ should be constructed for each different way p of parsing s by R.

Each member of the set $SI_s(V)$ is a list of lists containing the instances of a particular star variable:

$$\begin{array}{rll} \forall i \in [1, ns]: & SI_s(v_i) = (l_1^i \ ... \ l_{nlists(i)}^i) & \text{where } nlists(i) \geq 0 \\ \forall i \in [1, ns] \ \forall j \in [1, nlists(i)]: & l_j^i = (e_{j1}^i \ ... \ e_{j(nelems(i,j))}^i) & \text{where } nelems(i,j) \geq 1 \end{array}$$

The star instances stored in $SI_s(V)$ are organized according to the containment relationships described by the star tree $\mathcal{T}$. To this end, each list l_j^i is associated with two integer pointers *father_list*(l_j^i) and *father_elem*(l_j^i) that identify the instance of the father star variable from which the instances of v_i in l_j^i are derived. Fig. 8.2 shows an example of set of star instances, obtained for a given string and RE, where the corresponding star tree has four levels of star variables; for each list of instances, a pair of superindexes is given, where the first superindex denotes the *father_list* and the second one denotes the *father_elem*.

In general, for all the star variables that are in the *first level* of $\mathcal{T}$, the following structure arises:

$$\forall v_i: \ (r, v_i) \in \mathcal{T} \ \Rightarrow \ \begin{array}{l} SI_s(v_i) = (l_1^i) \ \wedge \ l_1^i = (e_{11}^i) \ \wedge \\ \mathit{father\_list}(l_1^i) = -1 \ \wedge \ \mathit{father\_elem}(l_1^i) = -1. \end{array}$$

For these variables, $nlists(i) = 1$, $nelems(i, 1) = 1$, and the *father_list* and *father_elem* pointers take an arbitrary null value; furthermore, if v_i is not instantiated in parsing s then $e_{11}^i = -1$ else $e_{11}^i \geq 0$ is the number of consecutive matches of the corresponding star operand in the only instance of v_i.

Otherwise, let v_f be the father of v_i in $\mathcal{T}$. For all the star variables that are in the *second level* of $\mathcal{T}$, the list of instance lists is either empty (when $e_{11}^f \leq 0$) or its structure is given by

$$\forall v_i: \ (r, v_f), (v_f, v_i) \in \mathcal{T} \ \wedge \ e_{11}^f > 0 \ \Rightarrow \ \begin{array}{l} SI_s(v_i) = (l_1^i) \ \wedge \ l_1^i = (e_{11}^i \ ... \ e_{1(e_{11}^f)}^i) \ \wedge \\ \mathit{father\_list}(l_1^i) = 1 \ \wedge \ \mathit{father\_elem}(l_1^i) = 1, \end{array}$$

i.e. $nlists(i) = 1$, $nelems(i, 1) = e_{11}^f$, and, if v_i is not instantiated in the k-th match of the star operand of v_f then $e_{1k}^i = -1$ else $e_{1k}^i \geq 0$ is the number of matches of the star operand of v_i in the k-th cycle.

$$
\begin{array}{lcl}
R & = & (a(b(ce^*c\ +\ df^*d)^*)^*)^* \\
V & = & \{v_1, v_2, v_3, v_4, v_5\} \\
R(V/*) & = & (a(b(ce^{v_1}c\ +\ df^{v_2}d)^{v_3})^{v_4})^{v_5} \\
\mathcal{T} & = & (V \cup r, \{(r,v_5),(v_5,v_4),(v_4,v_3),(v_3,v_1),(v_3,v_2)\}, r) \\
s & = & abccdffdcecbddbdfdceecabceeec \\
SI_s(v_5) & = & (\ (2)^{\{-1,-1\}}\ ) \\
SI_s(v_4) & = & (\ (3\ 1)^{\{1,1\}}\ ) \\
SI_s(v_3) & = & (\ (3\ 1\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}}\ ) \\
SI_s(v_1) & = & (\ (0\ \text{-}1\ 1)^{\{1,1\}}\ \ (\text{-}1)^{\{1,2\}}\ \ (\text{-}1\ 2)^{\{1,3\}}\ \ (3)^{\{2,1\}}\ ) \\
SI_s(v_2) & = & (\ (\text{-}1\ 2\ \text{-}1)^{\{1,1\}}\ \ (0)^{\{1,2\}}\ \ (1\ \text{-}1)^{\{1,3\}}\ \ (\text{-}1)^{\{2,1\}}\ )
\end{array}
$$

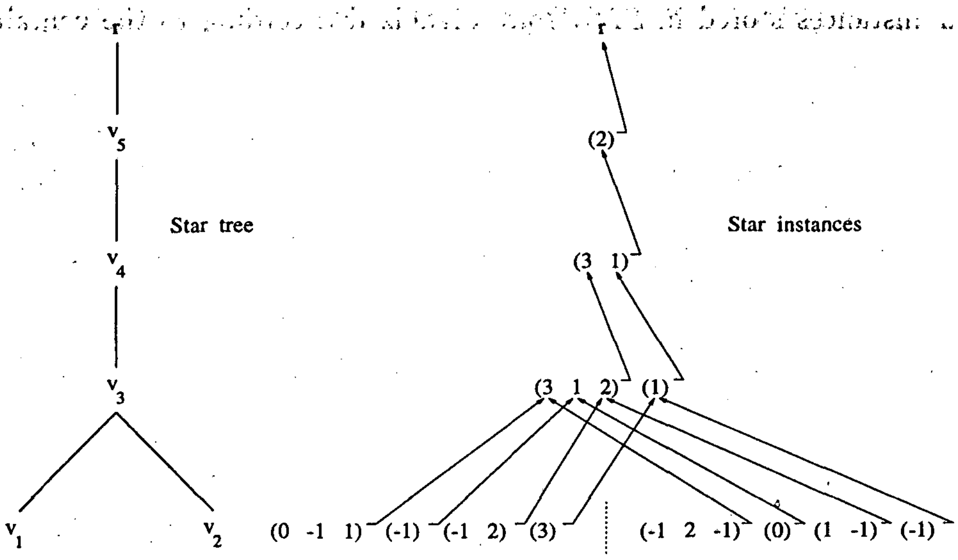


Fig. 8.2 *An example of star instances data structure.*

Finally, for all the star variables that are in the *second or higher levels* of $\mathcal{T}$, we have the following general rule for the instances of v_i with father v_f:

$$
nlists(i) = \#\{e^f_{j'k'}\ \mid\ e^f_{j'k'} > 0\}\quad \wedge\quad \forall j \in [1..nlists(i)]:\quad nelems(i,j) = e^f_{j'k'}\ \wedge
$$
$$
father\_list(l^i_j) = j'\ \wedge\ father\_elem(l^i_j) = k'
$$

and e^i_{jk} is either a natural value (if v_i is actually instantiated in the k-th cycle of the instance of v_f identified by the pointers {j',k'}) or -1 (if v_i is not actually instantiated in such cycle).

Two efficient algorithms for parsing strings by *unambiguous* REs that construct the *star instances* structure will be reported in Section 8.4.1; the first one just needs the RE and can always be applied, whereas the second one can only be applied if the RE has been obtained from a known DFA using the FSA-to-RE mapping proposed in Section 8.1 and it needs some inputs provided by that step. These RE parsing algorithms call some operations that deal with the star instances data structure, which are listed next together with their pre- and post-conditions. In what follows, $SI[i]$ is a computational representation of $SI_s(v_i)$, the star instances of v_i for a certain string s.

function *initialise_star_instances* $(V, \mathcal{T})$ **returns** star_instances
Precondition: V is a set of star variables and $\mathcal{T}$ is a star tree defined on V.
Postcondition: Let SI be the returned structure; $\forall i \in [1..|V|]$:
$v_i \in \text{first\_level}(\mathcal{T}) \Rightarrow SI[i] = ((-1)^{\{-1,-1\}}) \wedge$
$v_i \notin \text{first\_level}(\mathcal{T}) \Rightarrow SI[i] = ()$.

procedure *append_new_list_of_instances_to_son* (SI, i, j', k', m)
Precondition: $SI[i] = (l_1^i ... l_n^i)$ where $n = nlists(i) \geq 0$,
$\{j', k'\}$ identify an instance of the father v_f of v_i, and $m = e^f_{j'k'} > 0$.
Postcondition: $SI[i] = (l_1^i ... l_n^i l_{n+1}^i)$ where $nlists(i) = n + 1 \wedge nelems(i, n + 1) = m$ $\wedge$ $father\_list(l_{n+1}^i) = j' \wedge father\_elem(l_{n+1}^i) = k' \wedge \forall k \in [1..m] : e^i_{(n+1)k} = -1$.

procedure *assign_value_to_star_instance* $(SI, i, j, k, value)$
Precondition: $value \geq 0$ and $e^i_{jk} = -1$, where e^i_{jk} is the k-th element of the j-th list of instances in $SI[i]$.
Postcondition: $e^i_{jk} = value$.

function *read_value_of_star_instance* (SI, i, j, k) **returns** integer
Precondition: $e^i_{jk} = -1 \vee e^i_{jk} \geq 0$, where e^i_{jk} is the k-th element of the j-th list of instances in $SI[i]$.
Postcondition: e^i_{jk} is returned.

function *number_of_lists* (SI, i) **returns** integer
Precondition: $1 \leq i \leq |V|$, and $nlists(i) \geq 0$ is the number of lists of instances in $SI[i]$.
Postcondition: $nlists(i)$ is returned.

function *number_of_elements* (SI, i, j) **returns** integer
Precondition: $1 \leq i \leq |V|$, $1 \leq j \leq nlists(i)$, and $nelems(i, j) \geq 1$ is the number of elements in the j-th list of instances in $SI[i]$.
Postcondition: $nelems(i, j)$ is returned.

8.2.3 Definition of ARE and language described by an ARE

Definition 3.5. Given a set of star variables V, $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ is a set of *constraints* on the values (of the instances) of the star variables in V, which consists of a set of independent *linear equations* $\mathcal{L} = \{l_1, ..., l_{nd}\}$, where $0 \leq nd \leq |V|$, and a set of *bounds* $\mathcal{B} = \{b_1, ..., b_{ni}\}$, where $ni = |V| - nd$. The linear relations $\mathcal{L} = \{l_1, ..., l_{nd}\}$ partition the set V into two subsets V^{ind}, V^{dep} of independent and dependent star variables, respectively; this is

$$l_i \text{ is } v_i^{dep} = a_{i1}v_1^{ind} + .. + a_{ij}v_j^{ind} + .. + a_{i(ni)}v_{ni}^{ind} + a_{i0}, \quad \text{for } 1 \leq i \leq nd,$$

where ni and nd are the number of independent and dependent star variables, respectively, and $|V| = nd + ni$. The set of bounds $\mathcal{B} = \{b_1, ..., b_{ni}\}$ specifies a lower bound for each independent star variable; this is

$$b_j \text{ is } v_j^{ind} \geq c_j, \ c_j \in \mathcal{N}, \quad \text{for } 1 \leq j \leq ni.$$

The equations in $\mathcal{L}$ represent constraints that are only well-defined for natural values of the involved variables. Moreover, the coefficients a_{ij} of the linear relations will always be rational numbers. Likewise, the lower bounds c_j are always natural numbers, which, unless otherwise stated, are assumed to be zero by default.

To test the satisfaction of a set of linear equations $\mathcal{L}$ by a set of star instances, it is useful to rewrite each constraint $l_i \in \mathcal{L}$ by removing all the terms of independent variables with coefficient zero in the right hand sides of the equations; this is

$$l_i \text{ is } v_i^{dep} = a'_{i1}v_{i1} + .. + a'_{ik_i}v_{ik_i} + a_{i0}, \quad \text{for } 1 \leq i \leq nd,$$

where $\forall i \in [1, nd], j \in [1, k_i] : a'_{ij} \neq 0 \ \wedge \ v_{ij} \in V^{ind}$.

Definition 3.6. An *augmented regular expression* (or ARE for short) is a four-tupla $(R, V, \mathcal{T}, \mathcal{C})$, where R is a *regular expression* over an alphabet Σ, V is its associated set of *star variables*, $\mathcal{T}$ is its associated *star tree*, and $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ is a set of *constraints* formed by a set of independent *linear equations* $\mathcal{L} = \{l_1, ..., l_{nd}\}$ and a set of *bounds* $\mathcal{B} = \{b_1, ..., b_{ni}\}$, where $nd + ni = |V|$.

In order to define formally the language described by an ARE, some other definitions must be introduced previously, which are given next.

Definition 3.7. Given a star tree $\mathcal{T}$, a set of star instances $SI_s(V)$ for a certain string s, and two nodes $v_i, v_j \in V$, we say that v_i is a *degenerated ancestor* of v_j for s iff

i) v_i is an ancestor of v_j in $\mathcal{T}$, and
ii) for each instance of v_i in $SI_s(v_i)$, all the values of the instances of v_j in $SI_s(v_j)$ that are derived from it are constant.

The root r of $\mathcal{T}$ is considered, by definition, as a non-degenerated ancestor of any other node v_j for every string.

Definition 3.8. Let $v_i \in V \cup \{r\}$, $v_j \in V$; we say that v_i is the *housing ancestor* of v_j for a string s iff v_i is the *nearest* non-degenerated ancestor of v_j for s.

Definition 3.9. Let $\mathcal{T}$ be a star tree defined on V, and let $\mathcal{L}$ be a set of linear equations defined on V in which only the independent variables with non-zero coefficient appear in the right hand sides of the equations. Let $v_{c_i} \in V \cup \{r\}$ be the *deepest common ancestor* in $\mathcal{T}$ of the nodes $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$, which are labeled by the star variables involved in the equation $l_i \in \mathcal{L}$. Given a set of star instances $SI_s(V)$ for a certain string s, we say that $SI_s(V)$ *satisfies* an equation $l_i \in \mathcal{L}$ iff

i) the *housing ancestors* for s of the nodes $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$ are either v_{c_i} or an ancestor of v_{c_i}, or they satisfy a strict equality constraint, and
ii) the linear relation l_i is met by the values of the corresponding instances of $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$.

The first condition above implies structural similarity of instance lists, while the second one requires the satisfaction of the equation. The problem of constraint satisfaction is further discussed in Section 8.4.2, where an explanation of the above conditions is given.

Definition 3.10. A set of star instances $SI_s(V)$ *satisfies* a set of nd linear equations $\mathcal{L}$ iff $SI_s(V)$ satisfies every equation $l_i \in \mathcal{L}$, for $1 \leq i \leq nd$. Every set of star instances $SI_s(V)$ *satisfies* an empty set $\mathcal{L}$ with zero relations.

Definition 3.11. Given a set of star instances $SI_s(V)$ for a certain string s and a set of bounds $\mathcal{B}$, we say that $SI_s(V)$ *satisfies* a bound $b_j \in \mathcal{B}$, $v_j^{ind} \geq c_j$, iff the values of all the actual instances in $SI_s(v_j^{ind})$ are $\geq c_j$.

Definition 3.12. A set of star instances $SI_s(V)$ *satisfies* a set of ni bounds $\mathcal{B}$ iff $SI_s(V)$ satisfies every bound $b_j \in \mathcal{B}$, for $1 \leq j \leq ni$. Every set of star instances $SI_s(V)$ *satisfies* a set $\mathcal{B}$ in which all the lower bounds are zero (i.e. b_j is $v_j^{ind} \geq 0$, for $1 \leq j \leq ni$) and a set $\mathcal{B}$ with $ni = 0$ bounds[3].

[3]The latter is only possible if all the star variables are dependent and constant: $v_i^{dep} = a_{i0}$, for $1 \leq i \leq |V|$.

Definition 3.13. A set of star instances $SI_s(V)$ *satisfies* a set of constraints $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ iff $SI_s(V)$ satisfies $\mathcal{L}$ and $\mathcal{B}$.

An algorithm for constraint testing that evaluates the predicate *satisfies*$(SI_s(V), \mathcal{C})$ is described in Section 8.4.2.

Definition 3.14. Let $\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$ be an ARE over Σ. The *language* $L(\tilde{R})$ *represented by* $\tilde{R}$ is defined as $L(\tilde{R}) = \{s \in \Sigma^* \mid s \in L(R)$ and there exists a parsing of s by R in which the set of star instances $SI_s(V)$ *satisfies* $\mathcal{C}\}$.

8.3 Expressive power of AREs

The AREs permit to describe a class of context-sensitive languages by imposing a set of rules that constrain the language of a regular super-set. A very simple example is the language of rectangles $\{\ a^m b^n a^m b^n \mid m, n \geq 1\ \}$, which is well-known to be context-sensitive (see grammar G_1 in Fig. 8.3), and which is described by the ARE $\tilde{R}_1 = (R_1, V_1, \mathcal{T}_1, (\mathcal{L}_1, \mathcal{B}_1))$, where

$$\begin{aligned}
R_1 &= a^*b^*a^*b^* \\
V_1 &= \{v_1, v_2, v_3, v_4\} \\
R_1(V_1/*) &= a^{v_1}b^{v_2}a^{v_3}b^{v_4} \\
\mathcal{T}_1 &= (V_1 \cup r, \{(r, v_1), (r, v_2), (r, v_3), (r, v_4)\}, r) \\
\mathcal{L}_1 &= \{v_3 = v_1,\ v_4 = v_2\} \quad (V^{dep} = \{v_3, v_4\} \text{ and } nd = 2) \\
\mathcal{B}_1 &= \{v_1 \geq 1,\ v_2 \geq 1\} \quad (V^{ind} = \{v_1, v_2\} \text{ and } ni = 2).
\end{aligned}$$

Fig. 8.5 displays some pattern classes that can be represented by AREs. Furthermore, quite more complex languages with an arbitrary level of star embedment and multiple linear constraints, even among stars at different levels of embedment, can be described as well by the ARE formalism. Consider, for instance, the ARE $\tilde{R}_2 = (R_2, V_2, \mathcal{T}_2, (\mathcal{L}_2, \mathcal{B}_2))$ with

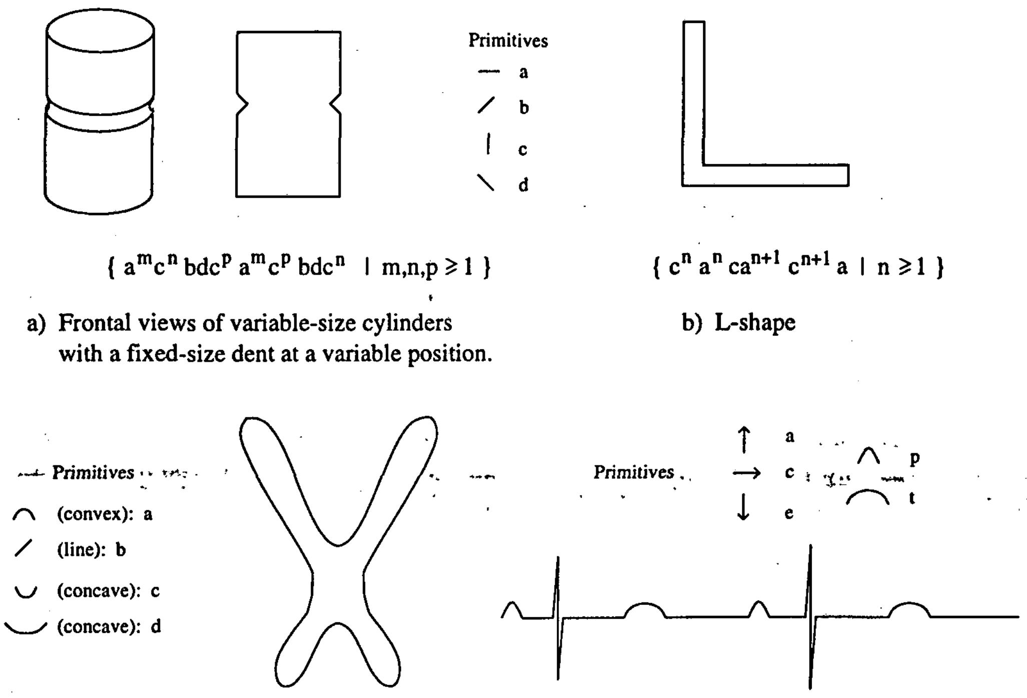
$$\begin{aligned}
R_2(V_2/*) &= (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^{v_{13}} \\
\mathcal{L}_2 &= \{v_{11} = v_1 + v_5 - v_7, \\
&\quad v_{12} = v_6, \\
&\quad v_2 = v_4 - 1, \\
&\quad v_3 = v_4 - 1, \\
&\quad v_8 = 0.5v_{10} + 0.5, \\
&\quad v_9 = 0.5v_{10} + 0.5\} \quad \text{and} \\
\mathcal{B}_2 &= \{v_4 \geq 2;\ v_6, v_{10}, v_{13} \geq 1;\ v_1, v_5, v_7 \geq 0\}.
\end{aligned}$$

1)	S	$\longrightarrow$	$abab$	2)	S	$\longrightarrow$	$aAbB[aC][bD]$	3)	A	$\longrightarrow$	aAE
4)	A	$\longrightarrow$	aF	5)	B	$\longrightarrow$	bBG	6)	B	$\longrightarrow$	bH
7)	$G[aC]$	$\longrightarrow$	$[aC]b$	8)	Gb	$\longrightarrow$	bb	9)	$G[bD]$	$\longrightarrow$	$b[bD]$
10)	$H[aC]$	$\longrightarrow$	$[aC]H$	11)	Hb	$\longrightarrow$	bH	12)	$H[bD]$	$\longrightarrow$	bb
13)	Eb	$\longrightarrow$	bE	14)	Ea	$\longrightarrow$	aa	15)	$E[aC]$	$\longrightarrow$	$a[aC]$
16)	Fb	$\longrightarrow$	bF	17)	Fa	$\longrightarrow$	aF	18)	$F[aC]$	$\longrightarrow$	aa

Fig. 8.3 *A grammar G_1 for the context-sensitive language* $\{ a^m b^n a^m b^n \mid m,n \geq 1 \}$.

1)	S	$\rightarrow$	AS	2)	S	$\rightarrow$	A	3)	A	$\rightarrow$	$EF[aG][eH]$
4)	F	$\rightarrow$	aFI	5)	F	$\rightarrow$	J	6)	$I[aG]$	$\rightarrow$	$[aG]I$
7)	$J[aG]$	$\rightarrow$	aG	8)	$I[eH]$	$\rightarrow$	$[eH]e$	9)	$G[eH]$	$\rightarrow$	Ge
10)	E	$\rightarrow$	$[LN]$	11)	E	$\rightarrow$	$[LN][cO]M$	12)	E	$\rightarrow$	$[cK]M[LO]$
13)	$[cK]$	$\rightarrow$	$c[cK]M$	14)	$[cK]$	$\rightarrow$	$[cN]$	15)	$[cO]$	$\rightarrow$	$c[cO]M$
16)	$[cO]$	$\rightarrow$	c	17)	$M[LO]$	$\rightarrow$	$[LO]M$	18)	$[cN][LO]$	$\rightarrow$	$c[LN]$
19)	$[LO]$	$\rightarrow$	$L[cO]M$	20)	$[LN]c$	$\rightarrow$	$L[cN]$	21)	$[cN]c$	$\rightarrow$	$c[cN]$
22)	$[cN]L$	$\rightarrow$	$c[LN]$	23)	Ma	$\rightarrow$	aM	24)	$[cN]a$	$\rightarrow$	$c[aN]$
25)	$[LN]a$	$\rightarrow$	$L[aN]$	26)	$[aN]a$	$\rightarrow$	$a[aN]$	27)	G	$\rightarrow$	$[PQR]$
28)	$M[PQR]$	$\rightarrow$	$M[QR]$	29)	$M[PQR]$	$\rightarrow$	$[cP][QR]$	30)	$[aN][PQR]$	$\rightarrow$	aQ
31)	$M[cP]$	$\rightarrow$	$[cP]M$	32)	$M[cP]$	$\rightarrow$	$[cP]c$	33)	Mc	$\rightarrow$	cM
34)	$[aN][cP]$	$\rightarrow$	$a[cN]$	35)	$[aN]Q$	$\rightarrow$	$a[QN]$	36)	$[cN][QR]$	$\rightarrow$	cQ
37)	$M[QR]$	$\rightarrow$	$Q[Rc]$	38)	MQ	$\rightarrow$	QM	39)	$M[Rc]$	$\rightarrow$	$[Rc]c$
40)	$[cN]Q$	$\rightarrow$	$c[QN]$	41)	$[QN][Rc]$	$\rightarrow$	Qc				
42)	L	$\rightarrow$	$[VT][LW]$	43)	$[LW]$	$\rightarrow$	$T[UW]$	44)	$[LW]$	$\rightarrow$	$TL[UW]$
45)	L	$\rightarrow$	TU	46)	L	$\rightarrow$	TLU	47)	TU	$\rightarrow$	TXb
48)	$T[UW]$	$\rightarrow$	$T[XW]b$	49)	bU	$\rightarrow$	Xbb	50)	bX	$\rightarrow$	Xb
51)	$b[UW]$	$\rightarrow$	$[XW]bb$	52)	$b[XW]$	$\rightarrow$	$[XW]b$	53)	TX	$\rightarrow$	Ud
54)	$T[XW]$	$\rightarrow$	$[UW]d$	55)	dX	$\rightarrow$	Udd	56)	dU	$\rightarrow$	Ud
57)	$d[XW]$	$\rightarrow$	$[UW]dd$	58)	$d[UW]$	$\rightarrow$	$[UW]d$	59)	$[VT]U$	$\rightarrow$	$[VX]b$
60)	$[VT][UW]$	$\rightarrow$	db	61)	$[VX]X$	$\rightarrow$	$[dV]X$	62)	$[VX][XW]$	$\rightarrow$	$[dV][XW]$
63)	$[dV]X$	$\rightarrow$	$d[dV]$	64)	$[dV][XW]$	$\rightarrow$	dd				
65)	Q	$\rightarrow$	$[VT']U'$	66)	Q	$\rightarrow$	$[VT']T'Q'U'$	67)	Q'	$\rightarrow$	$T'T'Q'U'$
68)	Q'	$\rightarrow$	$T'U'$	69)	$T'U'$	$\rightarrow$	$T'X'd$	70)	dU'	$\rightarrow$	$X'dd$
71)	dX'	$\rightarrow$	$X'd$	72)	$T'X'$	$\rightarrow$	$U'b$	73)	bX'	$\rightarrow$	$U'bb$
74)	bU'	$\rightarrow$	$U'b$	75)	$[VT']U'$	$\rightarrow$	$[VX']d$	76)	$[VX']X'$	$\rightarrow$	$[bV']X'$
77)	$[VX']d$	$\rightarrow$	bd	78)	$[bV']X'$	$\rightarrow$	$b[bV']$	79)	$[bV']d$	$\rightarrow$	bd

Fig. 8.4 *A grammar G_2 for the CSL* $\{ (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^+ \mid$
$v_{12} = v_6;\ v_{11} = v_1 + v_5 - v_7;\ v_2 = v_4 - 1;\ v_3 = v_4 - 1;\ v_8 = 0.5v_{10} + 0.5;$
$v_9 = 0.5v_{10} + 0.5;\ v_4 \geq 2;\ v_6, v_{10} \geq 1;\ v_1, v_5, v_7 \geq 0\}$, *where*
rules 3 – 9 *implement the constraint* $v_{12} = v_6$,
" 10 – 41 " $v_{11} = v_1 + v_5 - v_7$,
" 42 – 64 *implement the constraints* $v_2 = v_4 - 1,\ v_3 = v_4 - 1$,
" 65 – 79 " $v_8 = 0.5v_{10} + 0.5,\ v_9 = 0.5v_{10} + 0.5$.



$\{ a^m c^n bdc^p a^m c^p bdc^n \mid m,n,p \geq 1 \}$

a) Frontal views of variable-size cylinders with a fixed-size dent at a variable position.

$\{ c^n a^n ca^{n+1} c^{n+1} a \mid n \geq 1 \}$

b) L-shape

$\{ cb^m ab^m db^n ab^n cb^n ab^n db^m ab^m \mid n > m \}$

c) Submedian chromosomes

$\{ (pc^m a^n e^{2n} a^n c^{2m} tc^s)^+ \mid m,n,s \geq 1 \}$

d) ECG-like signals

Fig. 8.5 *Some pattern classes that can be described by AREs.*

The context-sensitive grammar G_2 in Fig. 8.4 generates $L(\tilde{R}_2)$. The reader is encouraged to compare the compact and descriptive representation provided by the ARE $\tilde{R}_2$ with the obscure grammar G_2, that comprises 79 rules. Fig. 8.7 shows an example that belongs to $L(\tilde{R}_2)$, given an alphabet of graphical primitives $\{\uparrow a, \nearrow b, \rightarrow c, \searrow d, \downarrow e\}$.

The following question naturally arises: *Can all the CSLs be represented by AREs?*

Theorem 8.1. *The augmented regular expressions do not describe all the context-sensitive languages.*

Proof. A counterexample is given by the CSL $\{a^k \mid k = 2^i \wedge i \geq 1\}$ [HoUl:79], which is generated by the context-sensitive grammar G_3 displayed in Fig. 8.6. This language is not describable because AREs can only filter the range of values of the star variables through linear relations and lower bounds, and the relations only involve the star variables but not any external variable (such as i in $L(G_3)$). Therefore, there is no ARE $\tilde{R} = (R, V, \mathcal{T}, (\mathcal{L}, \mathcal{B}))$ such that $\mathcal{L}$ can represent the constraint $v_1 = 2^i \wedge i \geq 1$ for $R(V/*) = a^{v_1}$. □

1)	S	$\rightarrow$	$[ACaB]$	2)	$[Ca]a$	$\rightarrow$	$aa[Ca]$	3)	$[Ca][aB]$	$\rightarrow$	$aa[CaB]$
4)	$[ACa]a$	$\rightarrow$	$[Aa]a[Ca]$	5)	$[ACa][aB]$	$\rightarrow$	$[Aa]a[CaB]$	6)	$[ACaB]$	$\rightarrow$	$[Aa][aCB]$
7)	$[CaB]$	$\rightarrow$	$a[aCB]$	8)	$[aCB]$	$\rightarrow$	$[aDB]$	9)	$[aCB]$	$\rightarrow$	$[aE]$
10)	$a[Da]$	$\rightarrow$	$[Da]a$	11)	$[aDB]$	$\rightarrow$	$[DaB]$	12)	$[Aa][Da]$	$\rightarrow$	$[ADa]a$
13)	$a[DaB]$	$\rightarrow$	$[Da][aB]$	14)	$[Aa][DaB]$	$\rightarrow$	$[ADa][aB]$	15)	$[ADa]$	$\rightarrow$	$[ACa]$
16)	$a[Ea]$	$\rightarrow$	$[Ea]a$	17)	$[aE]$	$\rightarrow$	$[Ea]$	18)	$[Aa][Ea]$	$\rightarrow$	$[AEa]a$
19)	$[AEa]$	$\rightarrow$	a								

Fig. 8.6 *A grammar G_3 for the context-sensitive language $\{ a^k \mid k = 2^i \wedge i \geq 1 \}$.*

The context-sensitive language $\{a^k \mid k \text{ is a prime } \}$ is another counterexample. Indeed, it seems reasonable to expect that a large class of CSLs will not be described by AREs either, due to the limited type of context constraints that can be represented. A still more negative result is stated in the following theorem.

Theorem 8.2. *The augmented regular expressions do not describe all the context-free languages.*

Proof. A counterexample is given by the even-linear language $L(G_{pal})$ that represents the set of palindromes over an alphabet $\Sigma = \{0, 1\}$, where $G_{pal} = (\{S\}, \{0,1\}, P, S)$ and $P = \{S \rightarrow \lambda,\ S \rightarrow 0,\ S \rightarrow 1,\ S \rightarrow 0S0,\ S \rightarrow 1S1\}$. An ARE that might be proposed to represent $L(G_{pal})$ would be $\tilde{R}_{pal} = (R_{pal}, V_{pal}, \mathcal{T}_{pal}, (\mathcal{L}_{pal}, \mathcal{B}_{pal}))$ with $R_{pal}(V_{pal}/*) = (0^{v_1}1)^{v_2}0^{v_3}(0 + 1 + \lambda)0^{v_4}(10^{v_5})^{v_6}$ and $\mathcal{L}_{pal} = \{v_4 = v_3,\ v_6 = v_2,\ v_5 = v_1\}$, but it turns out that $L(\tilde{R}_{pal}) \neq L(G_{pal})$, since the equation $v_5 = v_1$ is not correct and should be replaced by a constraint like "$v_5 = Reverse(v_1)$", that expressed the fact that the value of the first instance of v_1 must be equal to the value of the last instance of v_5, the value of the second instance of v_1 must be equal to the value of the penultimate instance of v_5, etc. Clearly, this kind of constraint cannot be expressed by any ARE, according to the given definition of ARE. □

Consider now the language $\{xx \mid x \in (0+1)^+\}$ generated by the context-sensitive grammar G_p that was displayed in Fig. 3.4. $L(G_p)$ also corresponds to the *pattern language* $L(p)$ defined by the one-variable *pattern* $p = xx$ over the binary alphabet $\Sigma = \{0, 1\}$, where the variable x stands for any string in Σ^+ [Angl:80b]. The ARE given by $(0+1)^{v_1}(0+1)^{v_2}$ with $\mathcal{L} = \{v_2 = v_1\}$ and $\mathcal{B} = \{v_1 \geq 1\}$ cannot express that the substrings associated with the instances of the operands of the stars denoted by v_1 and v_2 are identical. However, if the equivalence rule $(0+1)^* = (0^* + 1^*)^*$, Eq.(2.14), is applied before, then the ARE given by $(0^{v_1} + 1^{v_2})^{v_3}(0^{v_4} + 1^{v_5})^{v_6}$ with $\mathcal{L} = \{v_4 = v_1,\ v_5 = v_2,\ v_6 = v_3\}$ and $\mathcal{B} = \{v_1 \geq 1,\ v_2 \geq 1,\ v_3 \geq 1\}$ is able to describe the pattern language $L(p)$. A similar approach can be followed to represent any pattern language using an ARE, as demonstrated in the following theorem.

Theorem 8.3. *The class of languages described by augmented regular expressions properly contains the class of pattern languages.*

Proof. Let p be a pattern over (Σ, X), where $\Sigma = \{a_1, ..., a_m\}$ $(m \geq 2)$ is a set of constant symbols and $X = \{x_1, ..., x_k\}$ $(k \geq 0)$ is a set of variables. A variable x can be represented by an ARE $\tilde{R}_\Sigma = (R_\Sigma, V_\Sigma, \mathcal{T}_\Sigma, (\mathcal{L}_\Sigma, \mathcal{B}_\Sigma))$ given by $R_\Sigma(V_\Sigma/*) = (a_1^{v_1} + ... + a_m^{v_m})^{v_{m+1}}$ with $\mathcal{L}_\Sigma = \emptyset$ and $\mathcal{B}_\Sigma = \{v_1 \geq 1, ..., v_m \geq 1, v_{m+1} \geq 1\}$, which describes Σ^+. Let $t(i)$ be the number of occurrences of the variable x_i in the pattern p, for $1 \leq i \leq k$. Each occurrence x_{ij} of each variable x_i in p can be associated in principle with a duplicate of $\tilde{R}_\Sigma$ containing new and different star variables v_{ijl}, for $1 \leq i \leq k$, $1 \leq j \leq t(i)$, $1 \leq l \leq (m+1)$. Then, an ARE $\tilde{R}_p = (R_p, V_p, \mathcal{T}_p, (\mathcal{L}_p, \mathcal{B}_p))$ describing the pattern language defined by p can be stated where the underlying RE is $R_p = p[R_\Sigma/x_i]_{i=1}^k$ and the set of constraints is given by $\mathcal{L}_p = \{v_{ijl} = v_{i1l} \mid 1 \leq i \leq k,\ 2 \leq j \leq t(i),\ 1 \leq l \leq m+1\}$ and $\mathcal{B}_p = \{v_{i1l} \geq 1 \mid 1 \leq i \leq k,\ 1 \leq l \leq m+1\}$.

On the other hand, it is obvious that the class of pattern languages does not cover the class of languages represented by AREs. For example, the language of rectangles $L(\tilde{R}_1)$ and the CFL $\{0^{v_1}1^{v_2}0^{v_3} \mid v_2 = v_1 + v_3\}$ cannot be represented by any pattern language. □

Corollary 1. *For every pattern p over (Σ, X) with k variables in X and m constant symbols in Σ, an ARE $\tilde{R}_p$ that is equivalent to p can be constructed whose size is in $O(m \cdot |p|)$.*

Proof. Let $|p| = n + \sum_{i=1}^{k} t(i)$ be the length of p, where n is the number of constant symbols in p and $t(i)$ is the number of occurrences of variable x_i in p, for $1 \leq i \leq k$. The size of the equivalent ARE $\tilde{R}_p = (R_p, V_p, \mathcal{T}_p, (\mathcal{L}_p, \mathcal{B}_p))$ constructed in the proof of Theorem 8.3 is in $O(m \cdot |p|)$, since $|R_p| = n + (3m+2)(|p| - n)$, $|V_p| = (m+1)(|p| - n)$, $|\mathcal{L}_p| = (m+1)(|p| - n - k)$ and $|\mathcal{B}_p| = (m+1)k$. □

8.4 String recognition through AREs

Given an ARE $\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$, the recognition of a string s as belonging to the language $L(\tilde{R})$ can be clearly divided in two steps: parsing s by the underlying RE R, and if success, checking the satisfaction of constraints $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ by the star instances $SI_s(V)$ that result from the parsing. If the RE R is *unambiguous*, then a unique set of star instances $SI_s(V)$ is possible for each $s \in L(R)$, and therefore a single satisfaction problem must be analysed to test whether $s \in L(\tilde{R})$.

8.4.1 Parsing strings by REs to build the star instances structure

Two methods for *unambiguous* RE parsing are described in the next subsections which, given a string s and an unambiguous RE R, respond whether $s \in L(R)$ or not, and in the first case, build the corresponding set of star instances $SI_s(V)$. The processing of the input string in both methods is divided in two phases: the *recognition* and *construction* phases.

The first method, described by Algorithm 8.3, uses the RE R itself for *recognition*, and the *construction* phase is a kind of re-run of the recognition phase in which it is known in advance that the string will be successfully parsed by the RE, and thus, the true instances of the star variables can be recorded. To this end, the current star variable that is involved in parsing is tracked, and the value of each new instance is computed by counting the number of consecutive matches of the operand of the related star. The time complexity of the first method is $O(|s| \cdot |R|)$ globally and for both phases.

The second method, described by Algorithm 8.4, is a more efficient parsing method that can be run if the unambiguous RE R has been obtained from an equivalent DFA A by applying the FSA-to-RE mapping proposed in Section 8.1. This will be the case if the RE has been inferred from examples using an RGI algorithm that returns a DFA (see Chapter 9). This second parsing method uses, besides the DFA A, some of the REs α_{ij}^{l} yielded by (the modified) Algorithm 8.1 and the *skeleton* of R formed in the subsequent simplifying step, which has been explained in Section 8.1.3. The key point is that A (instead of R) is used for *recognition* in $O(|s|)$, and that the path of visited states guides the *construction* of the star instances structure $SI_s(V)$. There are two achievements that permit to reduce the time complexity of the *construction* phase too: the former is to locate the substrings of s that are associated with the cycles of the involved star-type REs by finding subpaths of visited states that start and end with the same state without passing through it; the latter is to select directly the term of the involved union-type REs that actually matches the corresponding substring without the need of attempting to parse the non-matched terms. Hence, the second method has a time complexity of $O(max\{|skel(R)|,\ n \cdot |s|\})$, due to the *construction* phase, where n is the number of states of A, $|s|$ and $|skel(R)|$ denote the lengths of the input string and the skeleton of R, respectively, and $|skel(R)| \leq |R|$.

8.4.1.1 An algorithm to determine the star instances based on parsing by an RE alone

Algorithm 8.3 is intended to recognize whether a given string s belongs to the language described by a given unambiguous RE R and, if it belongs, to determine the corresponding set of star instances $SI_s(V)$. In the *recognition* phase, only the input string and the RE are involved; whereas in the *construction* phase, the star variables V, the star tree $\mathcal{T}$, and the set of star instances $SI_s(V)$ being built are involved as well. The set of star variables V and the star tree $\mathcal{T}$ associated with the RE R are assumed to be inputs of the algorithm; recall that they can be determined from the RE using Algorithm 8.2.

Algorithm 8.3 uses three basic functions (*concatenation_parsing*, *union_parsing* and *star_parsing*) for matching the three types of REs that can appear as subexpressions within the given RE. These functions are recursive, since they call each other to parse shorter and shorter substrings. A concatenation-type RE is regarded as a sequence of factors, where each factor is either an alphabet symbol, a union-type RE or a star-type RE. A union-type RE is a sequence of (mutually exclusive) terms, where each one of them is a concatenation-type RE. Finally, a star-type RE is either an alphabet symbol or a union-type RE affected by a star symbol, which means that the star operand can be matched consecutively zero or more times. The worst-time complexity of Algorithm 8.3 is $O(|s| \cdot |R|)$, where the factor $|R|$ is due to the need of trying to parse sometimes a substring by the different terms of a union-type RE.

It can be noted that the three parsing functions receive a boolean argument, *build_SI*, that indicates whether to build or not the star instances structure while parsing. This allows the determination of the star instances without the need of backtracking and deleting instances, since the argument *build_SI* will only be TRUE when the involved substring has been successfully parsed by the same subexpression previously (with *build_SI*=FALSE) and it is known that the whole string s has been successfully parsed by the whole RE R. Hence, in the main body of the algorithm a call to *concatenation_parsing* with *build_SI*=FALSE is firstly performed to *recognize* s by R, and if the parsing is successful then a new call to the same function with *build_SI*=TRUE is carried out to *construct* the lists of star instances.

In the *construction* phase (when *build_SI*=TRUE), the star instances structure SI constitutes an input/output argument of the three parsing functions, and a node of the star tree, referred to as the "current father", is implicitly followed at every step. This node corresponds to the star variable associated with the star-type RE currently involved in parsing, if any, or to the root of $\mathcal{T}$, otherwise. An input argument of the parsing functions, *father_cycle*, is used to track the cycle of the current instance of the current father and to indicate, for all the star variables that are sons of the current father, which is the element of the current list of instances to which a value must be assigned next. If the root node r is the current father, then $father\_cycle = 1$ by definition.

In the case of parsing by a star-type RE, the maximum number of consecutive matches of the star operand is always attempted, and in the case of parsing by a union-type RE, the first term that matches the substring is always selected. In this way, Algorithm 8.3 works properly when the given RE R is unambiguous, but if R is ambiguous, then it should be noted that the algorithm only attempts one of the several possible ways of parsing the string and the result in that case is not meaningful (i.e. the parsing might fail for a string in $L(R)$). Therefore, another algorithm should be used for parsing ambiguous REs, which should be capable of exploring all the parsing possibilities and building a new set of star instances $SI^p_s(V)$ for each successful way p of parsing s by R.

ALGORITHM 8.3: ***Parses a string by an RE alone and builds the lists of instances of the associated star variables if the parsing is successful.***

Inputs:

R	is a given unambiguous RE over an alphabet Σ;
V	is the set of star variables associated with R, that includes the star position information required for the functions *pos* and pos^{-1} and the address of the corresponding node in the star tree $\mathcal{T}$;
$\mathcal{T}$	is the star tree associated with R;
s	is a given string from Σ^*;

Outputs:

parsed	is a boolean value that will be TRUE if s is successfully parsed by R and FALSE otherwise;
SI	is the set of lists of instances of the star variables in V resulting from the parsing of s by R (only when *parsed* =TRUE).

```
begin_algorithm
{ Recognition phase }
build_SI :=FALSE;  { build_SI is a flag used to signal whether the star instances must be
     computed in the next parsing function to be called }
{ Try to parse s by R, considering R as a concatenation of REs }
parsed := concatenation_parsing (s, length(s), R, length(R), 0, V, T, build_SI, 1, SI,
                                  parsed_length);
```

```
if parsed and parsed_length = length(s) then    { Construction phase }
        SI := initialise_star_instances (V, T);    { initialises SI by creating one list of one
                dummy instance for each star variable $v_j$ in the first level of $\mathcal{T}$ }
        build_SI :=TRUE;
        father_cycle := 1;    { where r is implicitly considered as the current father }
        { Parse again s by R, now building the structure of star instances SI }
        parsed := concatenation_parsing (s, length(s), R, length(R), 0, V, T, build_SI,
                                        father_cycle, SI, parsed_length);
else
        parsed :=FALSE;
end_if
end_algorithm
```

```
function concatenation_parsing (s, slength, P, Plength, p0, V, T, build_SI,
                        father_cycle, SI, parsed_length) returns boolean
input arguments:
        s              is a string of length slength;
        P              is an RE over Σ of length Plength;
        p0             is the position of P in the whole RE R;
        V and T        are the star variables and star tree associated with R, respectively;
        build_SI       tells whether to build SI or not while parsing;
        father_cycle   indicates the position of the next instance to be assigned in the
                       current list of instances of the star variables that are sons of the
                       current father (only relevant if build_SI=TRUE);
input/output arguments:
        SI   is updated with the star instances resulting from the parsing of s by the RE P
             (only when build_SI=TRUE);
output arguments:
        parsed_length   is the length of the longest prefix of s that is parsed by the RE P;
returned value:
        if $P = \lambda$ then       TRUE is returned iff $s = \lambda$
        else $\{P \neq \lambda\}$   TRUE is returned iff there is a prefix of s that is parsed by P;
begin
if  $P = \lambda$ then
        parsed_length := 0;
        return (slength = 0);
else
```

```
    $U$ := find_factors($P$);   { $U$ is a list of triples ($< u_1, p_1, l_1 > ... < u_m, p_m, l_m >$) where
        the $u_i$'s are REs such that $P = u_1 ... u_m$, $m \geq 1$, $p_i$ is the position of the RE $u_i$
        in $P$, and $l_i$ is the length of $u_i$ }
    $k := 0$;   { $k$ is the length of the prefix of $s$ already parsed }
    reset_list($U$);   $parsed$ :=TRUE;
    while $parsed$ and not end_of_list($U$) do
        $< u, p, l >$:= get_current_element($U$);
        if is_a_terminal_symbol($u$)  { $u = a$ for some $a \in \Sigma$ }  then
            $parsed := (s[k] = P[p])$;
            $k := k + 1$;
        elseif is_a_simple_startype_RE($u$)   { $u = a^*$ for some $a \in \Sigma$ }  then
            $value := 0$;   $a := P[p]$;
            while  $s[k] = a$ do
                $k := k + 1$;   $value := value + 1$;
            end_while
            if $build\_SI$ then
                $pp := p0 + p + 1$;   { is the position of $*$ in $R$ }
                $j$ := pos$^{-1}(V, pp)$;  { is the index of the associated star variable $v_j$
                    }
                $current\_list$ := number_of_lists ($SI, j$);
                $current\_elem$ := $father\_cycle$;   { $current\_list$ and $current\_elem$
                    point to the instance where to assign the value for $v_j$ }
                assign_value_to_star_instance ($SI, j, current\_list, current\_elem, value$);
            end_if
        elseif is_a_uniontype_RE($u$)   { $u = (P')$ where $(P')$ is an RE over $\Sigma$ }  then
            $substring$ := string_elements_from_to ($s, k, slength$);
            { Try to parse $substring$ by $u$, a union-type RE }
            $parsed$ := union_parsing ($substring, slength-k, u, l, p0+p, V, \mathcal{T}, build\_SI,$
                                    $father\_cycle, SI, substring\_parsed\_length$);
            $k := k + substring\_parsed\_length$;
        elseif is_a_parenthesised_startype_RE($u$)   { $u = (P')^*$  }  then
            $substring$ := string_elements_from_to ($s, k, slength$);
            { Parse $substring$ by $u$, a star-type RE }
            $parsed$ := star_parsing ($substring, slength - k, u, l, p0 + p, V, \mathcal{T}, build\_SI,$
                                    $father\_cycle, SI, substring\_parsed\_length$);
            $k := k + substring\_parsed\_length$;
        end_if
        move_to_next_element($U$);
    end_while
    if $parsed$ then  $parsed\_length := k$;
    else              $parsed\_length := 0$;
    end_if
    return $parsed$;
end_if

end_function
```

```
function union_parsing ($s, slength, P, Plength, p0, V, \mathcal{T}, build\_SI,$
                        $father\_cycle, SI, parsed\_length$) returns boolean
input arguments:
        $s$              is a string of length $slength$;
        $P$              is an RE of length $Plength$ and form $(P')$ over $\Sigma$;
        $p0$             is the position of $P$ in the whole RE $R$;
        $V$ and $\mathcal{T}$      are the star variables and star tree associated with $R$, respectively;
        $build\_SI$      tells whether to build $SI$ or not while parsing;
        $father\_cycle$  indicates the position of the next instance to be assigned in the
                         current list of instances of the star variables that are sons of the
                         current father (only relevant if $build\_SI$=TRUE);
input/output arguments:
        $SI$  is updated with the star instances resulting from the parsing of $s$ by the RE $P$
              (only when $build\_SI$=TRUE);
output arguments:
        $parsed\_length$  is the length of the longest prefix of $s$ that is parsed by the RE $P$;
returned value:
        TRUE   if there is a prefix of $s$ that is parsed by the RE $P$, and
        FALSE  otherwise;
begin
$T$ := find_terms($P$);   { $T$ is a list of triples $(< t_1, p_1, l_1 > ... < t_n, p_n, l_n >)$ where the $t_i$'s
    are REs such that $P = (t_1 + ... + t_n)$, $n \geq 1$, $p_i$ is the position of the RE $t_i$ in $P$, and
    $l_i$ is the length of $t_i$ }
reset_list($T$);   $parsed$ :=FALSE;
while not $parsed$ and not end_of_list($T$) do
    $< t, p, l >$:= get_current_element($T$);
    { Try to parse $s$ by $t$, considering $t$ as a concatenation-type RE }
    $parsed$ := concatenation_parsing ($s, slength, t, l, p0 + p, V, \mathcal{T}$,FALSE,
                                      $father\_cycle, SI, k$);
    if $parsed$ then   { the parsed term is the current $t$ }
        if $build\_SI$ then
            { Parse again $s$ by $t$ now computing the star instances }
            $parsed$ := concatenation_parsing ($s, slength, t, l, p0 + p, V, \mathcal{T}$,TRUE,
                                              $father\_cycle, SI, k$);
        end_if
    end_if
    move_to_next_element($T$);
end_while
if $parsed$ then  $parsed\_length := k$;
else              $parsed\_length := 0$;
end_if
return $parsed$;
end_function
```

```
function star_parsing ($s, slength, P, Plength, p0, V, \mathcal{T}, build\_SI,$
                       $father\_cycle, SI, parsed\_length$) returns boolean
input arguments:
        $s$              is a string of length $slength$;
        $P$              is an RE of length $Plength$ and form $(P')^*$ over $\Sigma$;
        $p0$             is the position of $P$ in the whole RE $R$;
        $V$ and $\mathcal{T}$      are the star variables and star tree associated with $R$, respectively;
        $build\_SI$      tells whether to build $SI$ or not while parsing;
        $father\_cycle$  indicates the position of the next instance to be assigned in the
                         current list of instances of the star variables that are sons of the
                         current father (only relevant if $build\_SI$=TRUE);
input/output arguments:
        $SI$  is updated with the star instances resulting from the parsing of $s$ by the RE $P$
              (only when $build\_SI$=TRUE);
output arguments:
        $parsed\_length$  is the length of the longest prefix of $s$ that is parsed by the RE $P$;
returned value:
        TRUE  is always returned, since at least $(P')^0$ is matched;
begin
$L$ := create_empty_list_of_strings();
$k$ := 0;   { $k$ is the length of the prefix of $s$ already parsed }
$parsed$ := TRUE; $i$ := 0;  { $i$ is the number of matches of $(P')$ found }
while $parsed$ and $k < slength$ do
    $substring$ := string_elements_from_to ($s, k, slength$);
    { Try to parse $substring$ by $(P')$, a union-type RE }
    $parsed$ := union_parsing ($substring, slength - k, (P'), Plength - 1, p0, V, \mathcal{T}$,FALSE,
                               $father\_cycle, SI, substring\_parsed\_length$);
    if $parsed$ then
        $i := i + 1$;
        $substring$ := string_elements_from_to ($s, k, k + substring\_parsed\_length$);
        $L$ := put_element_in_list_of_strings($L, substring$);
        $k := k + substring\_parsed\_length$;
    end_if
end_while
$parsed\_length := k$;
$value := i$;

if $build\_SI$ then
    $pp := p0 + Plength - 1$;  { is the position of $*$ in $R$ }
    $j := \text{pos}^{-1}(V, pp)$;  { is the index of the associated star variable $v_j$ }
    $current\_list$ := number_of_lists($SI, j$);
    $current\_elem := father\_cycle$;  { $current\_list$ and $current\_elem$ point to the instance
        where to assign the value for $v_j$ }
    assign_value_to_star_instance ($SI, j, current\_list, current\_elem, value$);
    if $value > 0$ then
        $father$ := startree_node_with_given_identifier ($\mathcal{T}, V, j$);  { $v_j$ is now the current
            father }
        $father\_list$ := $current\_list$;
        $father\_elem$ := $current\_elem$;
        { $father\_list$ and $father\_elem$ point to the instance of the current father in
            $SI$ that generates a new list of instances for its sons }
        $father\_value$ := $value$;  { $father\_value$ determines the length of the new list
            of instances to be created for the sons of the current father }
        for $k := 1$ to nsons_of ($\mathcal{T}, father$) do
            $node$ := k-th_son_of ($\mathcal{T}, father, k$);
            $son\_id$ := node_identifier_of ($node$);
            { Create a new list of dummy instances for $v_{son\_id}$, with as much elements
                as $father\_value$ }
            append_new_list_of_instances_to_son ($SI, son\_id, father\_list,$
                                                  $father\_elem, father\_value$);
        end_for
        reset_list($L$);
        for $i := 1$ to $father\_value$ do
            $substring$ := get_element_from_list_of_strings($L$);
            $l$ := length($substring$);
            $new\_father\_cycle$ := $i$;
            { Parse again $substring$ by ($P'$), a union-type RE, now computing the
                star instances }
            $parsed$ := union_parsing ($substring, l, (P'), Plength - 1, p0, V, \mathcal{T}$,TRUE,
                                  $new\_father\_cycle, SI, substring\_parsed\_length$);
            move_to_next_element($L$);
        end_for
    end_if
end_if
return TRUE;
end_function
```

8.4.1.2 *An algorithm to determine the star instances based on parsing by an RE with the help of an equivalent DFA*

A parsing method for unambiguous REs more efficient than the previous one is attainable if the given RE R has been obtained from an equivalent DFA $A = (Q, \Sigma, \delta, q_0, F)$ using the FSA-to-RE mapping proposed in Section 8.1. Algorithm 8.4 describes such a method, which, in addition to the source DFA, requires some other data provided by the FSA-to-RE transformation.

In this case, the *recognition* phase is carried out in $O(|s|)$ time, where $|s|$ is the length of the given string, using the DFA A, because $L(A) = L(R)$, and recording the path of visited states. When a positive string $s \in L(A)$ is recognized, then, in the *construction* phase, the path of visited states is split in a certain number of subpaths, which depend on the final state arrived and on the last visit to the states with lower index. This process can be regarded as finding an instantiation of the *skeleton* of R that matches the path of visited states. Remember that the skeleton of an RE R (over Σ) is an RE over a set of special symbols {"R0", "R0-1",...,"Ri-j",...,"R($f_{|F|}-1$)-$f_{|F|}$"}, $1 \leq j \leq f_{|F|}$, $0 \leq i < j$, that describes R in a synthetic way, where "R0" refers to the RE $R_0 = \alpha_{00}^{1}{}^*$, "Ri-j" refers to the RE $R_{ij} = \alpha_{ij}^{j}$, and $q_{f_{|F|}} \in F$ is the final state with the largest index within the ordered set of states Q.

We say that a subpath of states is matched to R_{ij} if the first element in the subpath corresponds to the last visit of q_i in the whole path, all the visits to states $q_k < q_i$ have been done previously, and the last element in the subpath corresponds to the last visit of q_j in the whole path. The subpaths of visited states that are matched to the REs $R_{ij} = \alpha_{ij}^{j}$ are further split in two parts, which correspond respectively to a union-type RE ($R'_{ij} = \alpha_{ij}^{j+1}$) and a star-type RE ($C_j = \alpha_{jj}^{j+1*}$); these two parts are separated by the first visit to q_j in the subpath. The subpath of states formed from the beginning of the whole path to the last visit of the initial state q_0 is matched to R_0. If $R_0 \neq \lambda$, the subpath matched to R_0 correspond to a star-type RE $C_0 = \alpha_{00}^{1}{}^*$.

Hence, the input string s can be split in substrings according to the path segmentation, using the length of each subpath. In this way, each substring so formed is parsed by the corresponding segment of the RE (either a union-type or a star-type RE) and, furthermore, the parsing process is guided by the associated subpath of visited states. To describe how the guiding mechanism works, the REs R'_{ij} and C_j, $0 \leq j \leq (n-1)$, $0 \leq i < j$, where n is the number of states of A, must be decomposed recursively in terms of the shorter REs from which Algorithm 8.1 has built them. By analysing Algorithm 8.1, the following relations can be established:

$$R'_{ij} = \alpha_{ij}^{j+1} = \alpha_{ij}^{n} + \alpha_{i(n-1)}^{n}\alpha_{(n-1)(n-1)}^{n}{}^*\alpha_{(n-1)j}^{n} + \cdots + \alpha_{i(j+1)}^{j+2}\alpha_{(j+1)(j+1)}^{j+2}{}^*\alpha_{(j+1)j}^{j+2}$$

$$C_j = \alpha_{jj}^{j+1*} = (\alpha_{jj}^{n} + \alpha_{j(n-1)}^{n}\alpha_{(n-1)(n-1)}^{n}{}^*\alpha_{(n-1)j}^{n} + \cdots + \alpha_{j(j+1)}^{j+2}\alpha_{(j+1)(j+1)}^{j+2}{}^*\alpha_{(j+1)j}^{j+2})^*$$

where some of the terms in these REs may represent the empty language $\emptyset$ for a given DFA A. Symmetrically, a similar relation can be established for the REs $R''_{ji} = \alpha^{j+1}_{ji}$, $1 \leq j \leq (n-1)$, $0 \leq i < j$:

$$R''_{ji} = \alpha^{j+1}_{ji} \;=\; \alpha^n_{ji} + \alpha^n_{j(n-1)}\alpha^n_{(n-1)(n-1)}{}^*\alpha^n_{(n-1)i} + \dots + \alpha^{j+2}_{j(j+1)}\alpha^{j+2}_{(j+1)(j+1)}{}^*\alpha^{j+2}_{(j+1)i}$$

For clarity purposes, the three preceding relations can be rewritten as follows:

$$R'_{ij} = \alpha^n_{ij} + R'_{i(n-1)}C_{n-1}R''_{(n-1)j} + \dots + R'_{i(j+1)}C_{j+1}R''_{(j+1)j} \quad (8.8)$$

$$C_j = (\ \alpha^n_{jj} + R'_{j(n-1)}C_{n-1}R''_{(n-1)j} + \dots + R'_{j(j+1)}C_{j+1}R''_{(j+1)j}\ )^* \quad (8.9)$$

$$R''_{ji} = \alpha^n_{ji} + R'_{j(n-1)}C_{n-1}R''_{(n-1)i} + \dots + R'_{j(j+1)}C_{j+1}R''_{(j+1)i} \quad (8.10)$$

Consequently, from all the REs α^l_{ij} $(1 \leq l \leq n,\ 0 \leq j < n,\ 0 \leq i < l)$ yielded by Algorithm 8.1, just a subset of at most $2{\cdot}n^2$ REs are needed to parse R efficiently, which correspond to the REs denoted by R'_{ij}, R''_{ji}, α^n_{ij}, α^n_{ji} (for $1 \leq j \leq (n-1)$, $0 \leq i < j$), C_j and α^n_{jj} (for $0 \leq j \leq (n-1)$).

Let us explain now how the parsing proceeds when matching a certain substring to a union-type RE of the form R'_{ij} (a similar argument applies to the R''_{ji} REs and to the union-type REs included in the C_j REs). Let the subpath of states associated with the substring be $p = p_0 ... p_r$, where we know that $p_0 = i$, $p_r = j$ and $\forall t \in [1, r-1] : (p_t > i) \wedge (p_t > j)$. If $r = 1$ then the term α^n_{ij} is selected as the matched term. Otherwise, the subpath must be scanned to find the state with the lowest index; let h be such a state, and let $z1$ and $z2$ be the positions of the first and last occurrence of h in the subpath p. Then, it is clear that the term $R'_{ih}C_hR''_{hj}$ is the matched term of R'_{ij}, and moreover, the three substrings associated with the subpaths $p_0...p_{z1}$, $p_{z1}...p_{z2}$ and $p_{z2}...p_r$ must be matched to the REs R'_{ih}, C_h and R''_{hj}, respectively (if $z1 = z2$ then the second substring is empty, and the star-type RE C_h is instantiated zero times).

Since the terms representing empty languages will not appear in the union-type REs included in R, an indexing mechanism is required to access directly to the selected term $R'_{ih}C_hR''_{hj}$ of R'_{ij}, given the state h. For example, an array of n elements, indexed by states, can contain the relative position of the terms that are actually present or a flag denoting absence. Such an array is needed for each one of the non-empty REs R'_{ij}, C_j and R''_{ji}, and it will be referred to as the index of the corresponding subexpression. Although it is not shown here, Algorithm 8.1 can be modified easily to compute and store these indices.

Concerning the process of matching a certain substring to a star-type RE of the form C_j, the first step is to split the associated subpath according to the occurrences of its first state. In this way, the substrings that are matched in each cycle of a star instance can be identified quickly. Then each of these substrings must be parsed by the RE that is the operand of the star in C_j (i.e. a union-type RE).

Algorithm 8.4 uses two guided parsing functions (*guided_union_parsing* and *guided_star_parsing*), which implement the irrevocable parsing strategy that has been described. The worst-case time complexity of the *construction* phase is $O(max\{|skel(R)|, n \cdot |s|\})$, where $|s|$ and $|skel(R)|$ denote the lengths of the input string and the skeleton of R, respectively. The cost $n \cdot |s|$ arises from the repeated scanning of subpaths in the search of the state with lowest index (less than n scans for each element of the global path of length $|s|$). The cost $|skel(R)|$ is due to the highest-level segmentation of the input string s, that involves scanning the skeleton of R. Since the *recognition* phase can be run in linear $O(|s|)$ time, the above complexity of the construction phase is also the one for the entire Algorithm 8.4.

ALGORITHM 8.4: *If an input string is accepted by a given DFA, the string is then parsed by a given equivalent RE, using the path of visited states and building the lists of instances of the associated star variables.*

Inputs:

A	is a given DFA $(Q, \Sigma, \delta, q_0, F)$ containing n states;
R	is the unambiguous RE obtained from A by applying Algorithm 8.1 and the simplification step given by Eq.(8.7);
$skelR$	is the skeleton of R found in the simplification step;
R'_{ij}, C_j, R''_{ji},	for $0 \leq j \leq n-1$, $0 \leq i < j$, are REs given by Algorithm 8.1;
α^n_{ij}, α^n_{jj}, α^n_{ji},	for $0 \leq j \leq n-1$, $0 \leq i < j$, are REs given by Algorithm 8.1;
IR'_{ij}, IC_j, IR''_{ji},	for $0 \leq j \leq n-1$, $0 \leq i < j$, are the indices of the terms in the REs R'_{ij}, C_j, R''_{ji}, respectively;
V	is the set of star variables associated with R, that includes the star position information required for the functions *pos* and pos^{-1} and the address of the corresponding node in the star tree $\mathcal{T}$;
$\mathcal{T}$	is the star tree associated with R;
s	is a given string from Σ^*;

Outputs:

parsed	is a boolean value that will be TRUE if s is accepted by A and FALSE otherwise;
path	is the path of states visited by A during the recognition of s;
skelR_instance	is the instance of the skeleton of R resulting from parsing s by R (only when *parsed* =TRUE);
SI	is the set of lists of instances of the star variables in V resulting from parsing s by R (only when *parsed* =TRUE);

```
begin_algorithm
{ Recognition phase }
parsed := recognize $(s, A, path)$;   { path is an output argument of this function }
if parsed then   { Construction phase }
    slength := length(s);
    last_state := path[slength];   { last_state is the index of the final state arrived in
                                     the recognition of s }
    { Find the last visit in path of each state lower than $q_{last\_state}$ such that $q_{last\_state}$ is
        reachable from it }
    $Q' := \{q_{last\_state}\}$;   { $Q' \subseteq F$ is a set whose only member is $q_{last\_state}$ }
    $G := C(Q')$;  { $G \subseteq Q$ is the transitive closure of $Q'$ with respect to the relation $\sim$:
                     $q_i \sim q_j \Leftrightarrow (j < i) \wedge R_{ji} \neq \emptyset$ }
    for $i := 0$ to $n - 1$ do
        if $q_i \in G$ then
            last_visit[i] := find_last_occurrence_of_state_in_path (i, path);
        end_if
    end_for
    { Initialise SI by creating one list of one dummy instance for each star variable $v_j$ in
        the first level of $\mathcal{T}$ }
    SI := initialise_star_instances $(V, \mathcal{T})$;
    father_cycle := 1;   { where r is implicitly considered as the current father }
    { Parse s by R guided by path and skelR }
    p0 := 0;   { p0 is the length of the part of R already scanned }
    if $C_0 \neq \lambda$ then
        substring := string_elements_from_to (s, 0, last_visit[0] − 1);  {$q_0 \in G$ always}
        subpath := path_elements_from_to (path, 0, last_visit[0]);
        l := length(substring);
        { Parse substring by $C_0$, a star-type RE, using subpath and computing the star
            instances }
        guided_star_parsing (substring, subpath, l, $C_0$, length($C_0$), p0, V, $\mathcal{T}$,
                             father_cycle, SI);
    end_if
    p0 := p0 + length($C_0$);
    skelR_instance :="R0";   { since $R_0 = C_0$ }
    psk0 := length("R0");   { psk0 is the length of the part of skelR already scanned }
```

```
        $i := 0$;
        while $i \neq last\_state$ do
            $j :=$ find_minimum_$q_j$_in_$G$_visited_later_than_$q_i$ $(i, G, last\_visit)$;
            $p0 :=$ find_next_Ri-j_in_skeleton $(i, j, skelR, R, psk0)$;   { scans $skelR$ from
                position $psk0$ to find the next occurrence of "Ri-j" and returns the position
                of next $R_{ij}$ in $R$ }
            $subpath :=$ path_elements_from_to $(path, last\_visit[i], last\_visit[j])$;
            $next\_visit\_j :=$ find_first_occurrence_of_j_in_subpath $(j, subpath)$;
            $substring\_1 :=$ string_elements_from_to $(s, last\_visit[i], next\_visit\_j - 1)$;
            $subpath\_1 :=$ path_elements_from_to $(path, last\_visit[i], next\_visit\_j)$;
            $l := length(substring\_1)$;
            { Parse $substring\_1$ by $R'_{ij}$, a union-type RE, using $subpath\_1$ and computing
                the star instances }
            guided_union_parsing $(substring\_1, subpath\_1, l, R'_{ij}, length(R'_{ij}), p0, V, \mathcal{T},$
                                    $father\_cycle, SI)$;
            if $C_j \neq \lambda$ then
                $substring\_2 :=$ string_elements_from_to $(s, next\_visit\_j, last\_visit[j] - 1)$;
                $subpath\_2 :=$ path_elements_from_to $(path, next\_visit\_j, last\_visit[j])$;
                $l := length(substring\_2)$;
                { Parse $substring\_2$ by $C_j$, a star-type RE, using $subpath\_2$ and
                    computing the star instances }
                guided_star_parsing $(substring\_2, subpath\_2, l, C_j, length(C_j),$
                                    $p0 + length(R'_{ij}), V, \mathcal{T}, father\_cycle, SI)$;
            end_if
            $p0 := p0 + length(R'_{ij}) + length(C_j)$;
            $skelR\_instance :=$ append_to_skeleton_instance $(skelR\_instance, $"Ri-j"$, i, j)$;
            $psk0 := psk0 + length($"Ri-j"$)$;   { $psk0$ is the length of the part of $skelR$
                already scanned }
            $i := j$;
        end_while
    end_if
end_algorithm
```

```
procedure guided_union_parsing ($s, path, slength, P, Plength, p0, V, \mathcal{T}, father\_cycle, SI$)
input arguments:
        $s$              is a string of length $slength$;
        $path$           is a state-index array of length $slength + 1$;
        $P$              is a union-type RE of length $Plength$ over $\Sigma$;
        $p0$             is the position of $P$ in the whole RE $R$;
        $V$ and $\mathcal{T}$      are the star variables and star tree associated with $R$, respectively;
        $father\_cycle$  indicates the position of the next instance to be assigned in the
                         current list of instances of the star variables that are sons of the
                         current father;
input/output arguments:
        $SI$  is updated with the star instances resulting from the parsing of $s$ by the RE $P$;
begin
$i := path[0]$;  $j := path[slength]$;
{ The RE $P$ is of the form $(t_1 + ... + t_k)$ where $k \geq 1$, and it corresponds to $R'_{ij}$ if $i < j$,
    $R''_{ij}$ if $i > j$, or $\alpha_{ii}^{i+1}$ if $i = j$, where $C_i = \alpha_{ii}^{i+1*}$ }
if  $slength = 1$  then
    { $t_1 = \alpha_{ij}^{n}$ is the matched term, and the star instances $SI$ are not affected. }
    { Do nothing }
else  { $slength > 1$ }
    $< h, z1, z2 >$:= find_lowest_state_in_subpath ($path, slength + 1$);  { always $h > i$ and
        $h > j$; this function also returns the position of the first and last occurrence of
        $h$ in $path$: $z1$ and $z2$, respectively }
    if  $i < j$  then
        $< m, p >$:= find_matched_term ($h, IR'_{ij}$);  { using the index of $R'_{ij}$ }
    elseif  $i > j$  then
        $< m, p >$:= find_matched_term ($h, IR''_{ij}$);  { using the index of $R''_{ij}$ }
    else  { $i = j$}
        $< m, p >$:= find_matched_term ($h, IC_i$);  { using the index of $C_i$ }
    end_if
    { $t_m = R'_{ih} C_h R''_{hj}$ is the matched term and $p$ is the position of $t_m$ in $P$ }
    $substring\_1$ := string_elements_from_to ($s, 0, z1 - 1$);
    $subpath\_1$ := path_elements_from_to ($path, 0, z1$);
    $l := length(substring\_1)$;
    { Parse $substring\_1$ by $R'_{ih}$, a union-type RE, using $subpath\_1$ and computing the
        star instances }
    guided_union_parsing ($substring\_1, subpath\_1, l, R'_{ih}, length(R'_{ih}), p0 + p, V, \mathcal{T}$,
                          $father\_cycle, SI$);
        $p := p + length(R'_{ih})$;
        if $C_h \neq \lambda$ then
            $substring\_2$ := string_elements_from_to $(s, z1, z2 - 1)$;
            $subpath\_2$ := path_elements_from_to $(path, z1, z2)$;
            $l := length(substring\_2)$;
            { Parse $substring\_2$ by $C_h$, a star-type RE, using $subpath\_2$ and computing the
                star instances }
            guided_star_parsing $(substring\_2, subpath\_2, l, C_h, length(C_h), p0 + p, V, \mathcal{T},$
                                    $father\_cycle, SI)$;
            $p := p + length(C_h)$;
        end_if
        $substring\_3$ := string_elements_from_to $(s, z2, slength - 1)$;
        $subpath\_3$ := path_elements_from_to $(path, z2, slength)$;
        $l := length(substring\_3)$;
        { Parse $substring\_3$ by $R''_{hj}$, a union-type RE, using $subpath\_3$ and computing the
            star instances }
        guided_union_parsing $(substring\_3, subpath\_3, l, R''_{hj}, length(R''_{hj}), p0 + p, V, \mathcal{T},$
                                $father\_cycle, SI)$;
    end_if
end_procedure
```

```
procedure guided_star_parsing $(s, path, slength, P, Plength, p0, V, \mathcal{T}, father\_cycle, SI)$
input arguments:
        $s$              is a string of length $slength$;
        $path$           is a state-index array of length $slength + 1$;
        $P$              is a star-type RE of length $Plength$ over $\Sigma$
                         (actually, $P = C_i = \alpha_{ii}^{i+1*}$, where $i = path[0]$);
        $p0$             is the position of $P$ in the whole RE $R$;
        $V$ and $\mathcal{T}$      are the star variables and star tree associated with $R$, respectively;
        $father\_cycle$  indicates the position of the next instance to be assigned in the
                         current list of instances of the star variables that are sons of the
                         current father;
input/output arguments:
        $SI$  is updated with the star instances resulting from the parsing of $s$ by the RE $P = C_i$ ;
begin
$i := path[0]$;  { and it always holds that $i = path[slength]$ }
```

```
$< S, e >$:= find_cycle_substrings($s, path, slength$); { $S$ is a list of triples ($< substr_1, p_1, l_1 >$,
    ..., $< substr_e, p_e, l_e >$), where $e \geq 0$, $s = substr_1...substr_e$ such that, for $1 \leq k \leq e$,
    $path[begin(substr_k)] = path[end(substr_k)] = i$ and $path[z] > i$ for $begin(substr_k) <$
    $z < end(substr_k)$, $p_k$ is the position of $substr_k$ in $s$, and $l_k$ is the length of $substr_k$ }
$value := e$;  { $e$ is the number of consecutive substrings of $s$ that are matched by $\alpha_{ii}^{i+1}$ }
{ Determine the star variable $v_j$ that has been instantiated }
$pp := p0 + Plength - 1$;  { is the position of $*$ in $R$ }
$j$ := pos$^{-1}$($V, pp$);  { is the index of the associated star variable $v_j$ }
{ Update the list of star instances of the star variable $v_j$ }
$current\_list$ := number_of_lists ($SI, j$);
$current\_elem := father\_cycle$;  { $current\_list$ and $current\_elem$ point to the instance
    where to assign the value for $v_j$ }
assign_value_to_star_instance ($SI, j, current\_list, current\_elem, value$);
if $value > 0$ then
    $father$ := startree_node_with_given_identifier ($\mathcal{T}, V, j$); {$v_j$ is now the current father}
    $father\_list := current\_list$;
    $father\_elem := current\_elem$;
    { $father\_list$ and $father\_elem$ point to the instance of the current father in $SI$ that
        generates a new list of instances for its sons }
    $father\_value := value$;  { $father\_value$ determines the length of the new list of
        instances to be created for the sons of the current father }
    for $k := 1$ to nsons_of ($\mathcal{T}, father$) do
        $node$ := k-th_son_of ($\mathcal{T}, father, k$);
        $son\_id$ := node_identifier_of ($node$);
        { Create a new list of dummy instances for $v_{son\_id}$, with as much elements as
            $father\_value$ }
        append_new_list_of_instances_to_son ($SI, son\_id, father\_list, father\_elem,$
                                              $father\_value$);
    end_for
    reset_list($S$);
    for $k := 1$ to $father\_value$ do
        $< substring, p, l >$:= get_current_element ($S$);
        $subpath$ := path_elements_from_to ($path, p, p + l$);
        $new\_father\_cycle := k$;
        { Parse $substring$ by $\alpha_{ii}^{i+1}$, a union-type RE, using $subpath$ and computing the
            star instances }
        guided_union_parsing ($substring, subpath, l, \alpha_{ii}^{i+1}, Plength - 1, p0, V, \mathcal{T},$
                              $new\_father\_cycle, SI$);
        move_to_next_element($S$);
    end_for
end_if
end_procedure
```

8.4.2 Testing the satisfaction of ARE constraints by star instances

Let $SI_s(V)$ be a set of star instances obtained from successfully parsing a certain string s by the underlying RE R of an ARE $\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$, where $\mathcal{C} = (\mathcal{L}, \mathcal{B})$. To determine whether $s \in L(\tilde{R})$ or not, the predicate $satisfies(SI_s(V), \mathcal{C})$ must be evaluated. The conditions under which a set of star instances $SI_s(V)$ *satisfies* a set of constraints $\mathcal{C}$ of an ARE have been formally established in Definitions 3.9 to 3.13 in Section 8.2.3. To sum up, $SI_s(V)$ *satisfies* $\mathcal{C}$ if and only if it fulfils all the linear relations $l_i \in \mathcal{L}$, for $1 \leq i \leq nd$, and all the bounds $b_j \in \mathcal{B}$, for $1 \leq j \leq ni$. A method to evaluate the predicate $satisfies(SI_s(V), \mathcal{C})$ is presented in Algorithm 8.5, whose theoretic time complexity is $O(|\mathcal{L}| \cdot |V| \cdot height(\mathcal{T}) \cdot I(SI_s(V)))$, where $I(SI_s(V)) = \max_{i=1,|V|} \sum_{j=1}^{nlists(i)} nelems(i,j)$ is the maximal number of instances of a star variable yielded by parsing s. A justification of this cost will be given after describing Algorithm 8.5.

In order to fulfil a linear relation l_i of an ARE, which is

$$v_i^{dep} = a'_{i1} v_{i1} + ... + a'_{ik_i} v_{ik_i} + a_{i0},$$

it is first required that the star variables involved in l_i, $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$, share a common structure of instances for the given string s. In other words, the total number of instances[4] of each one of these star variables should be the same in $SI_s(V)$, and the corresponding instance values should be grouped, one for each variable, in rows, one row for each cycle of the instances of a common ancestor.

At first, it would seem that the set of related star variables should be brothers in $\mathcal{T}$ and their father should be the common ancestor. However, if a constant value is found for the actual instances of a certain son v_j that are derived from an instance of its father v_i, and this occurs for all the father instances, then a unique instance value of the son may be associated with each instance value of the father, and therefore, regarding the structure $SI_s(V)$, the son may be promoted to a lower level in the tree of instance lists. In such a case, we say that the father v_i is a *degenerated ancestor* of its son v_j for s.

This promotion process may continue (towards the root of $\mathcal{T}$) until a *non-degenerated* ancestor v_h is found, called the *housing ancestor* of v_i for s, or a pre-established node of $\mathcal{T}$ is reached as housing ancestor by default. The housing ancestor by default can be a selected ancestor v_c that is shared with other star variables or the root node r of $\mathcal{T}$. Each time a star variable is promoted to a lower level in the tree of

[4]including *actual* and *dummy* instances

instance lists, all of its redundant instances are collapsed into a single one in order to fit in the same structure of instances of the degenerated ancestor. The procedure *determine_housing_ancestor_and_instances*, whose cost is $O(h(\mathcal{T}) \cdot I(SI_s(V)))$, just implements the promotion process described.

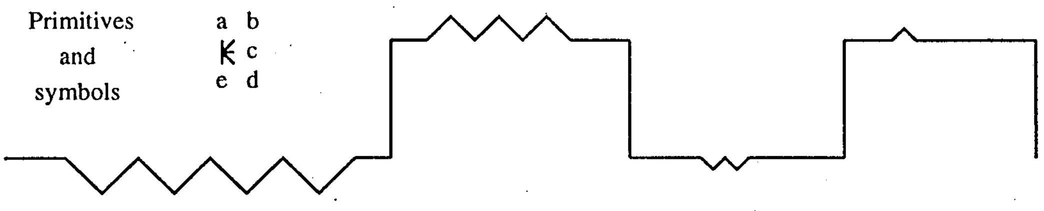
Moreover, even if a common housing ancestor is not found, a set of star variables $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$ might meet a linear relation l_i whenever all of their housing ancestors for a given string s satisfy a strict equality constraint. This fact ensures that a common structure of instances is available, even though the involved star variables cannot be promoted to an equivalent position (brothers) in the tree of instance lists, as it occurs in the AREs describing pattern languages. For example, in the ARE $(0^{v_1} + 1^{v_2})^{v_3}(0^{v_4} + 1^{v_5})^{v_6}$ with $\mathcal{L} = \{v_6 = v_3, v_4 = v_1, v_5 = v_2\}$ and $\mathcal{B} = \{v_3 \geq 1, v_1 \geq 1, v_2 \geq 1\}$, the relation $v_4 = v_1$ might be met by a set of star instances $SI_s(V)$ if v_6 and v_3 are the housing ancestors of v_4 and v_1, respectively, for s and the equality constraint $v_6 = v_3$ is previously demonstrated to be met.

In the case that, finally, the instances of the star variables $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$ can be arranged[5] in a two-dimensional array, one column for the instances of each star variable, one row for the instances corresponding to the same cycle, then the linear equation l_i can be actually checked on the instance values in each row of the array. To satisfy l_i, the linear relation must be met by all the rows.

On the other hand, to test a bound $b_j : v_j^{ind} \geq c_j$, it is only required to check whether all the actual instances in $SI_s(v_j^{ind})$ satisfy the given lower bound. Next, Algorithm 8.5 is described in more detail:

For each linear relation l_i in $\mathcal{L}$, the instances of the dependent star variable v_i^{dep}, stored in $SI_s(v_i^{dep})$, are analysed. If there is no actual instance of v_i^{dep}, the constraint l_i is considered to be met. Otherwise, the deepest common ancestor v_c of the star variables $\{v_i^{dep}, v_{i1}, ..., v_{ik_i}\}$, i.e. the first common ancestor going from each of these nodes to the root of $\mathcal{T}$, is selected as candidate to common housing ancestor. To verify the linear constraint l_i it is mandatory that the instances of all the independent star variables involved in the relation can be arranged in the structure of instances caused by the housing ancestor of v_i^{dep} (let us call it v_{hi}). Consequently, if any of them (say v_{ij}) has a housing ancestor (let us call it v_{hj}) that is deeper than v_c in $\mathcal{T}$ and the equation $v_{hj} = v_{hi}$ is not met by the instances, then it means that a shared structure of instances is not available for the string s, and therefore, the constraint l_i is considered to be violated. In the case of a constraint of the form $v_i^{dep} = a_{i0}$, no matter the level of v_i^{dep} in $\mathcal{T}$, its housing ancestor must be the root node and all the actual instances of v_i^{dep} must be collapsed to the constant value a_{i0} to verify the constraint.

[5](after some possible collapses of redundant instances)



$$
\begin{aligned}
s_1 &= c^5d^3b^3d^3b^3d^3b^3d^3b^3c^3a^{10}c^3b^2d^2b^2d^2b^2d^2c^5e^{10}c^6dbdbc^8a^{10}c^4bdc^{10}e^{10} \\
R_2(V_2/*) &= (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^{v_{13}}
\end{aligned}
$$

$$
\begin{array}{lcl lcl}
SI_{s_1}(v_{13}) & = & (\ (2)^{\{-1,-1\}}\ ) & & & \\
SI_{s_1}(v_1) & = & (\ (5\ 6)^{\{1,1\}}\ ) & SI_{s_1}(v_7) & = & (\ (3\ 4)^{\{1,1\}}\ ) \\
SI_{s_1}(v_5) & = & (\ (3\ 8)^{\{1,1\}}\ ) & SI_{s_1}(v_{11}) & = & (\ (5\ 10)^{\{1,1\}}\ ) \\
SI_{s_1}(v_6) & = & (\ (10\ 10)^{\{1,1\}}\ ) & SI_{s_1}(v_{12}) & = & (\ (10\ 10)^{\{1,1\}}\ ) \\
SI_{s_1}(v_4) & = & (\ (4\ 2)^{\{1,1\}}\ ) & SI_{s_1}(v_{10}) & = & (\ (3\ 1)^{\{1,1\}}\ ) \\
SI_{s_1}(v_2) & = & (\ (3\ 3\ 3\ 3)^{\{1,1\}}\ \ (1\ 1)^{\{1,2\}} & SI_{s_1}(v_8) & = & (\ (2\ 2\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}} \\
SI_{s_1}(v_3) & = & (\ (3\ 3\ 3\ 3)^{\{1,1\}}\ \ (1\ 1)^{\{1,2\}} & SI_{s_1}(v_9) & = & (\ (2\ 2\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}}
\end{array}
$$

Fig. 8.7 *An example of string (and associated pattern) recognized by the ARE $\tilde{R}_2$ and the set of star instances obtained during its parsing by the RE R_2.*

$$
\begin{array}{l}
\\
\text{1st cycle of instance } v_{13} = 2 \text{ in } s_1 \\
\text{2nd cycle of instance } v_{13} = 2 \text{ in } s_1
\end{array}
\quad
\left[\begin{array}{cccc}
 & v_1 & v_5 & v_7 \\
1 & 5 & 3 & 3 \\
1 & 6 & 8 & 4
\end{array}\right]
\left[\begin{array}{c}
0 \\ 1 \\ 1 \\ -1
\end{array}\right]
=
\left[\begin{array}{c}
v_{11} \\ 5 \\ 10
\end{array}\right]
$$

Fig. 8.8 *Verification of the constraint $v_{11} = v_1 + v_5 - v_7$ through the matrix product $A \cdot X = B$ (the top row of the displayed matrices A and B is just for labeling purposes).*

When the housing ancestor of all the star variables in l_i coincides with v_c or all the housing ancestors are related by strict equality, the constraint is tested on all the *actual instances* of v_i^{dep}. To this end, these instances are arranged in a column vector B, whereas the corresponding instances of the involved independent variables are orderly put as columns in a matrix A, together with an all-1's column associated with the constant term of the equation (see the procedure *build_system_matrix*). Then, it suffices to test $A \cdot X = B$, where X is the vector of coefficients in the right hand side of the equation l_i. The cost of the linear system construction and test is of $O(|V| \cdot I(SI_s(V)))$. If all the linear relations in $\mathcal{L}$ are satisfied in this way, then the bounds in $\mathcal{B}$ are checked on the actual instances of the corresponding independent star variables, until one of them is violated or all of them are successfully tested.

In the following pages, Algorithm 8.5 and their principal subprograms (*determine_housing_ancestor_and_instances*, *equal_instances* and *build_system_matrix*) are displayed. Note that *equal_instances* is a recursive function.

Consider the example of Fig. 8.7, where a string $s_1 \in L(\tilde{R}_2)$ and a pattern described by it are shown; the ARE $\tilde{R}_2 = (R_2, V_2, \mathcal{T}_2, (\mathcal{L}_2, \mathcal{B}_2))$ has been defined in Section 8.3. Given the constraints $\mathcal{C}_2 = (\mathcal{L}_2, \mathcal{B}_2)$ and the set of star instances displayed, which results from parsing s_1 by R_2, Algorithm 8.5 would set v_{13} as housed descendent of the root node r, and the rest of star variables in V_2 as housed descendents of v_{13}. In the two main loops of the algorithm, the six constraints in $\mathcal{L}_2$ and the seven bounds in $\mathcal{B}_2$ would be checked, respectively. The first linear relation, $v_{11} = v_1 + v_5 - v_7$, would lead to the successful test of the system $A \cdot X = B$ shown in Fig. 8.8. The rest of constraints in $\mathcal{L}_2$ would be verified similarly, and the satisfaction of the bounds in $\mathcal{B}_2$ would also be confirmed. Hence, the string s_1 of Fig. 8.7 would be accepted by Algorithm 8.5 as belonging to $L(\tilde{R}_2)$.

Now, let us discuss the complexity of Algorithm 8.5. The initialization step and the test of the bounds $\mathcal{B}$ have both a cost of $O(|V| \cdot I(SI_s(V)))$. The global cost of all the calls to the procedure *determine_housing_ancestor_and_instances* is $O(|V| \cdot height(\mathcal{T}) \cdot I(SI_s(V)))$, since at most one call for each star variable is performed, whereas the cost of all the calls to the procedure *build_system_matrix* and the function *test_linear_system* in the test of the relations $\mathcal{L}$ is $O(|\mathcal{L}| \cdot |V| \cdot I(SI_s(V)))$. Finally, the global cost of all the calls to the function *equal_instances*, in the worst-case sense, is $O(|\mathcal{L}| \cdot |V| \cdot height(\mathcal{T}) \cdot I(SI_s(V)))$, where the two former factors result from the number of calls to the function in the main loop of Algorithm 8.5, and the two latter factors come from the worst-case cost of the runs of the function *test_vector_equality* that may be carried out within a recursive chain of calls to *equal_instances*. Hence, the time complexity of Algorithm 8.5 is the sum of the above costs, of which the last one, $O(|\mathcal{L}| \cdot |V| \cdot height(\mathcal{T}) \cdot I(SI_s(V)))$, is the maximum (for a non-empty $\mathcal{L}$). On the other hand, the space complexity is $O(|V| \cdot I(SI_s(V)))$, which is related to the size of the set of star instances $SI_s(V)$.

If the function *equal_instances* were applied to each pair of star variables in V and the results stored prior to the test of $\mathcal{L}$, then the theoretical time complexity of Algorithm 8.5 might be somewhat disminished $O((|\mathcal{L}|+|V|+height(\mathcal{T})) \cdot |V| \cdot I(SI_s(V)))$ $= O(|V|^2 \cdot I(SI_s(V)))$, at the expense of introducing a storage requirement of $O(|V|^2)$. However, it is not clear whether this modification would improve or impair the average running time of the algorithm, and therefore, some empirical evidence of its benefits should be demonstrated.

ALGORITHM 8.5: *Evaluates the predicate satisfies($SI_s(V)$,$\mathcal{C}$), where $\mathcal{C} = (\mathcal{L}, \mathcal{B})$.*

Inputs:

V	is a set of star variables, that includes, for each variable, the address of the corresponding node in the star tree $\mathcal{T}$;
$\mathcal{T}$	is a star tree containing the variables in V as nodes;
SI	is a set of star instances $SI_s(V)$ resulting from parsing a string s by the RE from which V and $\mathcal{T}$ have been built;
$\mathcal{L}$	is a set of nd linear relations among the star variables in V;
$\mathcal{B}$	is a set of ni bounds defined on V;

Outputs:

satisfy_constraints	is a boolean value that will be TRUE if $SI_s(V)$ satisfies $(\mathcal{L}, \mathcal{B})$ and FALSE otherwise;

```
begin_algorithm
{ Initialization step }
$nstrings$ := 1;  $ASI[1]$ := $SI$;  { the set $SI$ is placed in the first position of an array of
    sets of star instances $ASI$ just for compatibility with the arguments of the functions
    determine_housing_ancestor_and_instances and sum_of_instance_values }
for $k := 1$ to $|V|$ do
    $housing\_ancestor\_of\_node[k] := -1$;  { marks that the housing ancestor of the star
        variable $v_k$ has not been computed yet }
    $node$ := startree_node_with_given_identifier ($\mathcal{T}, V, k$);
    if  nsons_of ($\mathcal{T}, node$) $> 0$ then
        $ninstances\_of\_housed\_descendents[k]$ := sum_of_instance_values ($ASI, nstrings, k$);
            { computes the sum of the values of all the actual instances in $SI_s(v_k)$ }
    end_if
end_for
$ninstances\_of\_housed\_descendents[0] := nstrings$;  { for the root node (identified by 0) }
{ Check the set of linear relations $\mathcal{L}$ }
reset_list($\mathcal{L}$);  $satisfy\_constraints$ :=TRUE;
while $satisfy\_constraints$ and not end_of_list($\mathcal{L}$) do   { check a constraint $l_i$ }
    < $vdep\_id, right\_hand\_side$ >:= get_current_element($\mathcal{L}$);
    $X$ := $right\_hand\_side.coefficients$;
    $list\_of\_indep\_variables$ := $right\_hand\_side.independent\_variables$;
    $common\_housing\_anc\_id$ := deepest_common_ancestor ($\mathcal{T}, V, vdep\_id$,
                                                    $list\_of\_indep\_variables$);
```

```
if housing_ancestor_of_node [vdep_id] < 0 then
    node := startree_node_with_given_identifier ($\mathcal{T}, V, vdep\_id$);
    determine_housing_ancestor_and_instances ($\mathcal{T}, node, vdep\_id, ASI, nstrings,$
        $common\_housing\_anc\_id, anc\_id, instances[vdep\_id]$);   { returns anc_id,
        the identifier of the housing ancestor of node, and instances[vdep_id], the
        values of the instances of $v_{vdep\_id}$ derived from the instances of $v_{anc\_id}$ in
        $ASI[1]$ }
    housing_ancestor_of_node [vdep_id] := anc_id;
end_if

if number_of_actual_instances (instances[vdep_id]) > 0 then
    exists_common_housing_ancestor := TRUE;
    if housing_ancestor_of_node [vdep_id] $\neq$ common_housing_anc_id then
        exists_common_housing_ancestor := FALSE;
    end_if
    reset_list (list_of_indep_variables);
    while satisfy_constraints and not end_of_list(list_of_indep_variables) do
        v_id := get_current_element (list_of_indep_variables);
        if housing_ancestor_of_node [v_id] < 0 then
            node := startree_node_with_given_identifier ($\mathcal{T}, V, v\_id$);
            determine_housing_ancestor_and_instances ($\mathcal{T}, node, v\_id, ASI, nstrings,$
                    $common\_housing\_anc\_id, anc\_id, instances[v\_id]$);
              { returns anc_id, the identifier of the housing ancestor of node,
              and instances[v_id], the values of the instances of $v_{v\_id}$ derived
              from the instances of $v_{anc\_id}$ in $ASI[1]$ }
            housing_ancestor_of_node [v_id] := anc_id;
        end_if
        if housing_ancestor_of_node [v_id] $\neq$ common_housing_anc_id then
            exists_common_housing_ancestor := FALSE;
        end_if
        if not exists_common_housing_ancestor then
            hi := housing_ancestor_of_node [vdep_id];
            hj := housing_ancestor_of_node [v_id];
            if not equal_instances ($hi, hj, ASI, nstrings, \mathcal{T}, V,$
              ninstances_of_housed_descendents, housing_ancestor_of_node,
              instances) then
                    satisfy_constraints := FALSE;
            end_if
        end_if
        move_to_next_element(list_of_indep_variables);
    end_while
```

```
            if $satisfy\_constraints$ then
                $anc\_id :=$ $housing\_ancestor\_of\_node$ [$vdep\_id$];
                $row\_max\_number :=$ $ninstances\_of\_housed\_descendents$ [$anc\_id$];
                $column\_number :=$ length($list\_of\_indep\_variables$) + 1;
                build_system_matrix ($vdep\_id, list\_of\_indep\_variables, instances,$
                        $row\_max\_number, column\_number, row\_actual\_number, B, A$);
                  { vector $B$ is given by the actual instances of $v_{vdep\_id}$, and the system
                  matrix $A$ is given by the star instances of the independent variables
                  in the r.h.s. of the linear equation, after removing the rows corre-
                  sponding to dummy instances of $v_{vdep\_id}$ }
                $satisfy\_constraints :=$ test_linear_system ($A, X, B, row\_actual\_number,$
                                                            $column\_number$);
            end_if
        end_if
        move_to_next_element($\mathcal{L}$);
    end_while
    { $satisfy\_constraints \equiv satisfies(SI_s(V), \mathcal{L})$ }

    { Check the set of bounds $\mathcal{B}$, if $satisfies(SI_s(V), \mathcal{L})$ }
    reset_list($\mathcal{B}$);
    while $satisfy\_constraints$ and not end_of_list($\mathcal{B}$) do  { check a bound $b_j$ : $v_j^{ind} \geq c_j$ }
        < $vind\_id, lower\_bound$ >:= get_current_element($\mathcal{B}$);
        $satisfy\_constraints :=$ test_lower_bound ($ASI, nstrings, vind\_id, lower\_bound$);
        move_to_next_element($\mathcal{B}$);
    end_while
    { $satisfy\_constraints \equiv satisfies(SI_s(V), \mathcal{C})$ }
    end_algorithm
```

procedure *determine_housing_ancestor_and_instances* ($\mathcal{T}$, *son*, *son_id*, *ASI*, *number_of_strings*, *default_housing_anc_id*, *anc_id*, *inst*)

input arguments:

$\mathcal{T}$	is the star tree containing the variables in V as nodes;
son	is a node of the star tree $\mathcal{T}$;
son_id	is the integer identifier of *son*;
ASI	is an array of sets of star instances of the variables in V;
number_of_strings	is the size of the array *ASI*;
default_housing_anc_id	is the identifier of a node of $\mathcal{T}$ to be chosen as housing ancestor of *son* when its nearer ancestors are degenerated;

output arguments:

anc_id	is the node identifier of the housing ancestor of *son*;
inst	is a vector containing the values of the "non-redundant" instances of the star variable v_{son_id} throughout the array *ASI*;

```
begin
node := son;   node_id := son_id;
current_ancestor := father_of ($\mathcal{T}$, node);
anc_id := node_identifier_of (current_ancestor);
current_ancestor_is_father :=TRUE;
found :=FALSE;

while not found do
    redundancy :=TRUE;
    i := 1;   i_prime := 1;
    for n := 1 to number_of_strings do
        nlists := number_of_lists (ASI[n], node_id);
        for j := 1 to nlists do
            nelements := number_of_elements (ASI[n], node_id, j);
            still_no_actual_instance_in_list :=TRUE;
            common_value := −1;
            for k := 1 to nelements do
                { j and k successively point to the instances of $v_{node\_id}$ in ASI[n] }
                value := read_value_of_star_instance (ASI[n], node_id, j, k);
                { determine the corresponding value in the vector inst }
                if current_ancestor_is_father then
                    inst_value := value;   { the assigned value comes directly
                        from ASI[n][son_id] }
                else   { current ancestor is not the father of son}
                    if  $value > 0$ then
                        { find the common value of the instances of $v_{son\_id}$ derived
                         from this instance of $v_{node\_id}$ }
                        $inst\_value := -1$;
                        for  $e := 1$ to  $value$ do
                            if  $inst\_value < 0$ and  $inst\_backup[i\_prime] \geq 0$ then
                                $inst\_value := inst\_backup[i\_prime]$;
                            end_if
                            $i\_prime := i\_prime + 1$;
                        end_for
                    else   { no instance of $v_{son\_id}$ is derived }
                        $inst\_value := -1$;
                    end_if
                end_if
                $inst[i] := inst\_value$;
                $i := i + 1$;
                { check the redundancy of the instances for the current list }
                if  $still\_no\_actual\_instance\_in\_list$ then
                    if  $inst\_value \geq 0$ then
                        $still\_no\_actual\_instance\_in\_list$ :=FALSE;
                        $common\_value := inst\_value$;
                    end_if
                else   {already found an actual instance in current list}
                    if  $inst\_value \geq 0$ and $inst\_value \neq common\_value$ then
                        $redundancy$ :=FALSE;
                    end_if
                end_if
            end_for
        end_for
    end_for

    if $redundancy$ and  $anc\_id \neq default\_housing\_anc\_id$ then
        { the current ancestor is degenerated }
        copy_integer_vector ($inst, inst\_backup, i - 1$);
        $node := current\_ancestor$;   $node\_id$ := node_identifier_of ($node$);
        $current\_ancestor$ := father_of ($\mathcal{T}, node$);
        $anc\_id$ := node_identifier_of ($current\_ancestor$);
        $current\_ancestor\_is\_father$ :=FALSE;
    else   { the housing ancestor of son has been determined }
        $found$ :=TRUE;
    end_if
end_while
end_procedure
```

function *equal_instances* $(left\_vid, right\_vid, ASI, number\_of\_strings, \mathcal{T}, V,$
$ninstances\_of\_housed\_descendents, housing\_ancestor\_of\_node, instances)$
returns boolean

input arguments:

$left\_vid$ and $right\_vid$	are the identifiers of two nodes of the star tree $\mathcal{T}$ (maybe the same);
ASI	is an array of sets of star instances (of V);
$number\_of\_strings$	is the size of the array ASI;
$\mathcal{T}$	is the star tree containing the variables in V as nodes;
V	is the set of star variables whose instances are in ASI;
$ninstances\_of\_housed\_descendents$	is an array that stores this number for each father node in $\mathcal{T}$;

input/output arguments:

$housing\_ancestor\_of\_node$	is an array containing the identifiers of the housing ancestors of each star variable;
$instances$	is an array containing the values of the "non-redundant" instances of each star variable throughout the array ASI;

returned value:

TRUE	if $left\_vid = right\_vid$ or there is a strict equality between the instances of the star variables v_{left_vid} and v_{right_vid} throughout the array ASI, and
FALSE	otherwise;

```
begin
if $left\_vid = right\_vid$ then $result$ :=TRUE;
else
    if $(left\_vid = 0 \wedge right\_vid \neq 0) \vee (left\_vid \neq 0 \wedge right\_vid = 0)$ then
        $result$ :=FALSE;
    else
        if $housing\_ancestor\_of\_node\ [left\_vid] < 0$ then
            $node$ := startree_node_with_given_identifier $(\mathcal{T}, V, left\_vid)$;
            determine_housing_ancestor_and_instances $(\mathcal{T}, node, left\_vid, ASI,$
                $number\_of\_strings, 0, left\_anc\_id, instances[left\_vid])$;
            $housing\_ancestor\_of\_node\ [left\_vid] := left\_anc\_id$;
        end_if
        if $housing\_ancestor\_of\_node\ [right\_vid] < 0$ then
            $node$ := startree_node_with_given_identifier $(\mathcal{T}, V, right\_vid)$;
            determine_housing_ancestor_and_instances $(\mathcal{T}, node, right\_vid, ASI,$
                $number\_of\_strings, 0, right\_anc\_id, instances[right\_vid])$;
            $housing\_ancestor\_of\_node\ [right\_vid] := right\_anc\_id$;
        end_if
```

```
            hl := housing_ancestor_of_node [left_vid];
            hr := housing_ancestor_of_node [right_vid];
            if  equal_instances (hl, hr, ASI, number_of_strings, T, V,
                  ninstances_of_housed_descendents, housing_ancestor_of_node, instances)
                then
                  dimension := ninstances_of_housed_descendents[hl];
                  result := test_vector_equality (instances[left_vid], instances[right_vid],
                                                  dimension);
            else
                  result :=FALSE;
            end_if
        end_if
    end_if
    return result;
    end_function
```

```
procedure build_system_matrix (star_id, list_of_indep_star_variables, instances,
              row_max_number, column_max_number, row_actual_number, B, A)
```

input arguments:

star_id	is the identifier of a star variable;
list_of_indep_star_variables	is a list of identifiers of star variables (v_k);
instances	is an array containing the values of the "non-redundant" instances of each star variable;
row_max_number	is the size of the array *instances*[*star_id*] and of each of the arrays *instances*[k] for all the v_k in *list_of_indep_star_variables*;
col_max_number	is the number of columns in the returned matrix A, which is *length*(*list_of_indep_star_variables*) + 1;

output arguments:

row_actual_number	is the number of rows in the returned matrix A and the number of elements in the returned vector B;
B	is a vector which contains the values of the "non-redundant" actual instances of the star variable v_{star_id};
A	is a matrix with a column of 1's and some more columns containing the instances of the star variables v_k in the given *list_of_indep_star_variables*;

```
begin    { of build_system_matrix }
nrow := 0;
for i := 1 to row_max_number do
    value := instances[star_id][i];
    { for each actual instance of star variable $v_{star\_id}$ write the corresponding row of
        instance values of the independent variables }
    if value $\geq$ 0 then
        nrow := nrow + 1;
        B[nrow] := value;
        A[nrow, 1] := 1;
        reset_list (list_of_indep_star_variables);
        for j := 2 to column_max_number do
            k := get_current_element (list_of_indep_star_variables);
            value := instances[k][i];
            if value < 0 then
                value := 0;   {convert dummy instances into zero-valued instances}
            end_if
            A[nrow, j] := value;
            move_to_next_element (list_of_indep_star_variables);
        end_for
    end_if
end_for
row_actual_number := nrow;
end_procedure
```