An Efficient Algorithm for Recognizing the Forward-Branching Class of Term-Rewriting Systems

Alain Miniussi, Robert Strandh *
Laboratoire Bordelais de Recherche en Informatique
Université Bordeaux 1, 351 cours de la Libération, 33405 Talence Cedex, France.

May 20, 1997

Abstract

We present an algorithm for recognizing the forward-branching (FB) [Str89] class of term-rewriting systems.

The algorithm is an improvement over a previous algorithm reported in [DS91] and [DS90] with a complexity proof in [Dur94]. The time complexity of the algorithm in [Dur94] is $O(\sum_i |n_i|^2)$ but was reported to be $O((\sum_i |n_i|)^2)$, and the space complexity is $O((\sum_i |n_i|)^2)$. We improve on this result by giving an algorithm with both time and space complexity $O(\sum_i |n_i|^2)$. Thus, in the normal case, where the size of each individual left-hand side is relatively small, the new algorithm is much more efficient. Since the quadratic behavior is caused by a set intersection operation that can be implemented with bit vectors, the algorithm is very efficient in practice.

Keywords: term-rewriting systems recognition, complexity, equational programming.

1 Introduction

In [HO82] Hoffmann and O'Donnell showed that recognizing the class of strongly left-sequential (SLS) term-rewriting systems can be done in linear time. Their method was based on the Aho-Corasick generalization [AC75] of the Knuth-Morris-Pratt [KMP77] pattern matching algorithm for words. In [KM91] Klop and Middledorp conjecture that recognizing the class of strongly sequential (SS) systems is NP-complete.

In [Str88] we defined the class of forward-branching (FB) systems and gave the first algorithm for recognizing it. Although the algorithm presented was able to decide whenever a given system was in FB, it was unable to construct the corresponding pattern-matching automaton. Furthermore, no complexity was given for the algorithm.

In [Dur94] I. Durand gave an algorithm that both recognizes FB and constructs the corresponding automaton. The complexity of that algorithm was shown to be quadratic

*{miniussi,strandh}@labri.u-bordeaux.fr, phone: +33 (0) 556 84 60 84, fax: +33 (0) 556 84 66 69

†Unité de Recherche Associée au Centre National de la Recherche Scientifique n.1304.

in the overall size of the system, where the size of the system is the sum of the sizes of the left-hand sides. In fact, the complexity was really the sum of the square of the size of each individual left-hand side, but this was not shown. Also, the space complexity was quadratic in the overall size of the system due to the use of a large table.

In this paper we eliminate the large table and get a time and space complexity that are both expressed as the sum of the square of the size of each individual left-hand side.

2 Terminology and Notation

2.1 Term Rewriting Sytems

We follow the notation of [HL79]. We also use some notation of [KM91] and [SR90].

Let $\mathcal{F}_n$ be a set of *function symbols* of arity n, $\mathcal{F} = \bigcup_{n\geq 0}\mathcal{F}_n$, and $\mathcal{V} \bigcap \mathcal{F} = \emptyset$ a denumerable set of *variable symbols*. Our expression language is the set $\mathcal{M}(\mathcal{F}, \mathcal{V})$ of first order *terms* formed from $\mathcal{F}$ and $\mathcal{V}$. We use $\mathcal{T}$ to denote $\mathcal{M}(\mathcal{F}, \mathcal{V})$ when $\mathcal{F}$ and $\mathcal{V}$ are fixed by the context.

Let M be a term, we use $\mathcal{O}(M)$ to denote its set of *occurrences*. An occurrence of M is a possibly empty sequence of positive integers such that:

$$\Lambda \in \mathcal{O}(M),$$

$$u \in \mathcal{O}(M_i) \Rightarrow iu \in \mathcal{O}(F(\cdots, M_i, \cdots)).$$

In the literature occurrences are also called *positions* or *paths*. $\overline{\mathcal{O}}(M)$ denotes the set of nonvariable occurrences in M. The set of occurrences is partially ordered by the *prefix ordering* $\leq$. We use $<$ to denote the strict ordering associated with $\leq$. We use M/u to denote the *subterm of M at u*, u/v to denote the path w such that $wv = u$ and $root(M)$ to denote the root symbol of M. Finally, if $u \in \mathcal{O}(M)$, $M[u \leftarrow N]$ is the *replacement in M at u by N*.

A *term rewriting system* (TRS for short) is a finite set Σ of pairs of terms $L_i \rightarrow R_i$ such that L_i is not a variable and $\mathcal{V}(R_i) \subseteq \mathcal{V}(L_i)$ (where $\mathcal{V}(M)$ is the set of variables of M). Red_Σ denotes the set of left-hand sides of Σ.

A *substitution* σ is a mapping from $\mathcal{T}$ to $\mathcal{T}$ satisfying $\sigma(F(M_1, \ldots, M_n)) = F(\sigma(M_1), \ldots, \sigma(M_n))$ (σ is determined by its restriction to $\mathcal{V}$). Any term M such that $M = \sigma(N)$ for some substitution σ and $N \in Red_\Sigma$ is called a *redex* of Σ. $\mathcal{R}_\Sigma(M)$ denotes the set of redex occurrences in M; a term is in *normal form* iff $\mathcal{R}_\Sigma(M) = \emptyset$.

From now on we will consider that Σ is fixed and drop the subscript Σ from $\mathcal{R}_\Sigma$.

2.2 Reduction

We say that the term M *reduces to N at occurrence u using rule $L_i \rightarrow R_i$* iff there exists a substitution σ such that $M/u = \sigma(L_i)$ and $N = M[u \leftarrow \sigma(R_i)]$. We write $M \rightarrow N$ when M reduces to N. We use $\stackrel{*}{\rightarrow}$ to denote the reflexive and transitive closure of $\rightarrow$.

Forward-branching programs are *orthogonal* (i.e., left linear and with no overlap [HL79],[KM91]). [Ros73] showed that for orthogonal TRSs, $\rightarrow$ is confluent.

We need to represent partial knowledge of a term. A symbol Ω of arity zero is thus introduced. It will represent our ignorance. We call *Ω-term* a term where an Ω can occur. $\mathcal{T}_\Omega$ will denote the set of all Ω-terms. Now we can define the *prefix ordering* $\preceq$ on $\mathcal{T}_\Omega$

where Ω is at the bottom. It is easy and straightforward to extend all the previously defined operations on terms to Ω-terms.

Along with Ω-terms and $\preceq$, we give some notation and definitions. We say that an Ω-term is in *Ω-normal form* iff $\mathcal{R}(N) = \emptyset$ and reserve the expression *normal form* for terms containing neither redexes nor Ω's. M_Ω denotes the term obtained from M by replacing its variables with Ω's and, if $F \in \mathcal{F}_n$, $F[\vec{\Omega}]$ denote the Ω-term $F(\Omega_1, \ldots, \Omega_n)$. If M is left-hand side[1], M_Ω is called a *redex scheme*, or simply a *scheme*. The set formed by the schemes is denoted Red_Ω. An Ω-term M is *preredex* iff $M \preceq N$ for some $N \in Red_\Omega$ and a *proper preredex* iff $M \prec N$. A proper subscheme whose root is labeled with a symbol appearing at the root of some scheme is called a *functional subscheme*, Let us call Red'_Ω the set Red_Ω augmented with all the *functional subschemes*.

Given an Ω-term M, $\mathcal{O}_\Omega(M)$ denotes the set of its Ω-occurrences (occurrences that correspond to an Ω) and $\overline{\mathcal{O}}_\Omega(M)$ the set of its non-Ω-occurrences.

2.3 Strong Index, Index Tree

Huet and Lévy [HL79] showed that given an orthogonal system, every term M not in normal form contains a *needed redex* (redex that needs to be replaced in every sequence of reductions leading to the normal form). However, for orthogonal systems, it is undecidable in general whether a redex is needed. To avoid that problem, one must use a more restricted class, for instance *Strongly Sequential* systems (SS for short). The following will be useful since $FB \subset SS$ [Str88].

The concept of sequentiality for a monotonic predicate was first introduced by [KP78].

Definition 1 *A predicate P on $\mathcal{T}_\Omega$ is* monotonic *iff $P(T)$ implies $P(T')$ whenever $T \preceq T'$.*

Let P be a monotonic predicate on $\mathcal{T}_\Omega$. An Ω-occurrence u of M is an index *with respect to P iff $\forall N$ such that $M \preceq N$, $P(N) = true$ implies $N/u \neq \Omega$.*

Then P is sequential at *M iff whenever $P(M) = false$, it follows that there exists an index with respect to P in M.*

Intuitively, an index is an Ω-occurrence that will need to be refined in order to make the predicate true, and sequentiality means that, for any Ω-term which does not satisfy the predicate, such an Ω-occurrence exists.

There is no need, in the scope of this paper, to use the normal reduction. In fact, we choose to ignore the right-hand sides by using only two kinds of reduction:

Notation 1 *Let M be an Ω-term.*

- *arbitrary reduction: $M \rightarrow_? M[u \leftarrow T]$ iff $u \in \mathcal{R}(M)$, there are no constraints on T.*
- *Ω-reduction: $M \longrightarrow_\Omega M[u \leftarrow \Omega]$ iff M/u is redex compatible (i.e., can be refined to a redex) and $u \in \overline{\mathcal{O}}_\Omega(M)$.*

We can now define the predicate $nf_?$. The fact that it is monotonic is easily checked.

Definition 2 *$nf_?(M) = true$ iff $\exists N$ in normal form such that $M \stackrel{*}{\rightarrow}_? N$.*

[1] old : a redex

From now, we will use *index* to denote an index with respect to $nf_?$, and $\mathcal{I}(M)$ will denote the set of indexes associated with M. From these definitions, strongly sequential means sequential according to $nf_?$ at any M in Ω-normal form. Intuitively, $u \in \mathcal{I}(M)$ cannot disappear without being refined.

Note that, whereas it is easy to decide whether an Ω-occurrence of an Ω-term is an index, deciding whether a TRS is strongly sequential is not a trivial matter; the first proof was given by [HL79]; another proof can be found in [KM91], where Klop and Middeldorp conjecture that deciding strong sequentiality is NP-complete although [Dur95] shows that the problem is in CO-NP.

We now give some definitions and terminology due to [O'D85] in order to define an *index tree*. An index tree is similar to the matching dag used by Huet and Lévy.

An Ω-term M is a *potential redex* if there is a way to refine it and then arbitrarily reduce it so that it becomes a redex. A potential redex is called a *soft term* in [KM91]. We say that $u \in \mathcal{O}(M)$ is a *potential redex occurrence* iff M/u is a potential redex. Symmetrically, we say that M is in *strong head normal form* iff it's not a potential redex and that $u \in \mathcal{O}(M)$ is a *strongly stable occurence* if M/u is in strong head normal form. Intuitively, the root term of an Ω-term in strong head normal form cannot change, even if M is refined and arbitrarily reduced.

Definition 3 *Let M be an Ω-term. M is a* firm *Ω-term iff $\exists u \in \mathcal{O}_\Omega(M)$ such that $\forall v \in \overline{\mathcal{O}}_\Omega(M)$, either v is strongly stable or $v < u$. We call such an occurrence u a* firm extension occurrence of M.

An index point *is a pair (M, u) where M is a firm Ω-term and also a preredex, u is a firm extension occurrence of M and $u \in \mathcal{I}(M)$.*

Definition 4 *Given an index point $s = (M, w)$ such that $M \neq \Omega$, an index point $t = (N, v)$ is a* failure point of (M, w) *iff $\exists u \neq \Lambda$ such that $w = uv$ and $N = M/u$. A failure point t of s is* the immediate failure point *of s iff every failure point of s other than t is a failure point of t; then we note $\Phi(s) = t$.*

From the implementation point of view, failure points can be used in order to not waste inspection time. If you find nothing interesting at M/w, you can still use the fact that you have inspected $N = M/u$. Actually, that property is not really useful for the forward-branching class since stabilization can be done before inspection [Str88].

We now give the definition for the index tree.

Definition 5 *An* index tree *is a pattern matching automaton whose states are index points and which, along with the usual transition function σ, has a* failure transition function Φ *that maps all the internal states to their immediate failure points.*

3 Characterisation of FB

We now give the definition of the forward-branching class. This definition, due to [Str88], is not really intuitive.

Definition 6 (Strandh 88) *An index tree is said to be a* forward-branching index tree *if and only if every state of the index tree can be reached from the root without following a failure transition.*

A TRS Σ is said to be forward-branching *if and only if a forward-branching index tree exists for Σ.*

We can characterize forward-branching more intuitively by the following property.

Property 1 *Every proper preredex M contains at least one Ω-occurrence which is not an Ω-occurrence of any scheme or subscheme greater than M.*

This property states that, for every state of the index tree, you know that there is a place to look at that does not correspond to a variable in a left-hand-side. We will call the class characterized by Property 1 K. Note that, from the programmer's point of view, this property is useful since it gives an intuitive idea of the kind of programs of that class.

Theorem 1 (I. Durand 91) *K=FB.*

Here is an example, which is a modification of one of the examples given in [Dur94] , that illustrates the theorem.

Example 1 *Consider the forward-branching system Γ:*

$K(F(G(x,C),A)) \rightarrow A$
$F(G(x,A),A) \rightarrow x$
$F(G(x,A),B) \rightarrow G(x,x)$
$G(B,B) \rightarrow B$

We have:

1. $Red_\Omega = \{1 : K(F(G(\Omega,C),A)),\ 2 : F(G(\Omega,A),A),\ 3 : F(G(\Omega,A),B),\ 4 : G(B,B)\}$ *and*
2. $Red'_\Omega = Red_\Omega \cup \{1_{/1} : F(G(\Omega,C),A),\ 1_{/1.1} : G(\Omega,C),\ 2_{/1} : G(\Omega,A),\ 3_{/1} : G(\Omega,A)\}$.

A forward-branching index tree for Γ is given in Figure 1, and you can easily check that Γ satisfies Property 1. The function trnsf, illustrated in the figure, can be ignored for the moment. The function Φ is represented with a dashed line. We do not represent the failure transitions when $\Phi(s) = root$.

Note that Γ is strongly ordered [Str88], a more restricted class than FB. Still, it is a small example that can be used to illustrate some interesting points in the scope of this paper.

The proof of Theorem 1 can be found in [Dur94]. The most important lemma used in the proof of $K = FB$ is probably the *construction lemma* [Dur94], that shows how to build a forward-branching index tree for some TRS Σ satisfying Property 1. The proof of that lemma is constructive; here is the algorithm that produces a forward-branching index tree associated with Σ.

Initialization:

let $nofinal_0 = \{(\Omega, \Lambda)\}, max = Max\{|M| \mid M \in Red_\Omega\}$
for all i, $0 < i < max$ **let** $nofinal_i = \emptyset$
end for

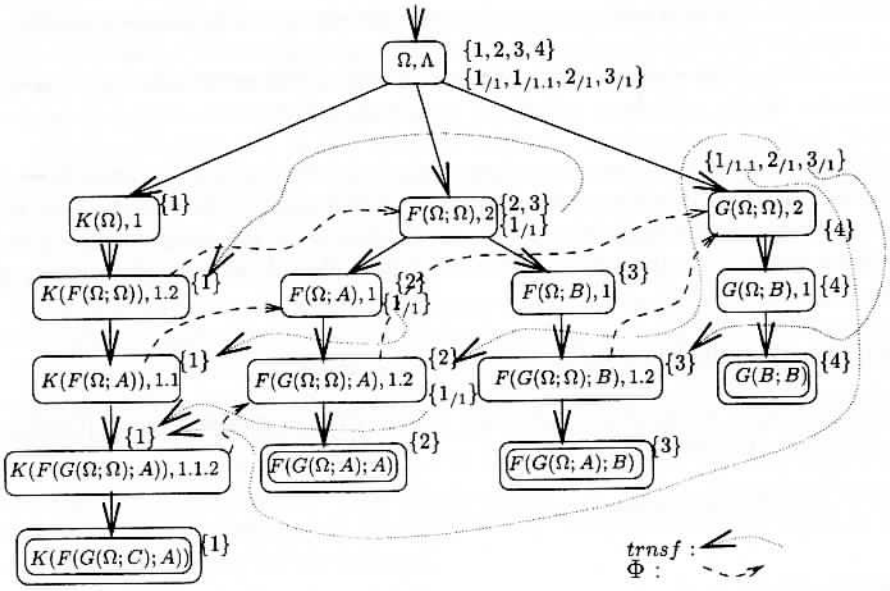

Figure 1: Forward-branching index tree for Γ

Induction step:

for all $s_n = (M_n, u_n) \in nofinal_n$,
  for all F occurring in Σ s.t. $\exists N \in Red_\Omega$ s.t. $M_n \prec N$
  and $root(N/u_n) = F$,
    let $M_{n+1} = M_n[u_n \leftarrow F[\vec{\Omega}]]$
    if $M_{n+1} \in Red_\Omega$ **let** $s_{n+1} = M_{n+1}$
    else let $s_{n+1} = (M_{n+1}, u_{n+1})$, $nofinal_{n+1} = nofinal_{n+1} \cup \{s_{n+1}\}$
      with u_{n+1} **chosen** as indicated below;
    end if
    let $\delta(s_n, F) = s_{n+1}$
  end for
end for

The choice of u_{n+1} is made as follows.
Consider the sequence of index points reachable from s_n via failure transitions only.

Case 1: None of these index points t is such that $\delta(t, F)$ is defined. Then we choose u_{n+1} so that $\forall N \in Red'_\Omega$ such that $M_{n+1} < N, N/u_{n+1} \neq \Omega$. This choice is always possible if Property 1 is satisfied.

Case 2: Such a t exists, $t = (U, v)$ and $\delta(t, F) = (U', v')$. By Definition 4, there is a u such that $u_n = uv$. We choose $u_{n+1} = uv'$.

Case 2 means that we try, whenever possible, to use the same inspection order for M_{n+1}/u as the one used for U'. This technique is useful since it allows us to reuse the inspection work done on U'.

4 General Algorithm

We first express the algorithm abstractly with set operations. We then show how to implement these set operations for an efficient result.

4.1 Data Structure

- A *term* is a structure containing the following fields:

  symbol, the symbol of the root of the term;
  child, a vector of terms of the size of the arity of *symbol*.

- A *prefix* is a structure that we use to encode a prefix of a redex scheme M together with a set of occurences of M (typically a set of indexes or a set of Ω-occurrences of M). The structure has two fields:

  terms, an array of the size of M, containing pointers to subterms of M;
  distinguished, a set of occurrences in the *terms* array.

  Note that, *terms* being coded with an array, *distinguished* can be coded by a bit vector: if the bit n is set, then $terms[n]$ is distinguished (see section 5).

- Finally, a *state* s of the automaton contains the following fields:

  scheme_prefixes, a set of prefixes of redex schemes;
  subscheme_prefixes, a set of prefixes of functional subschemes;
  indexes, a set of occurrences corresponding to possible indexes for s;
  index, the final choice of index for s;
  successors, a set of pairs $< f, t >$ $(\sigma(s, f) = t)$;
  psize, an offset that codes the first free place in the *terms* array of the prefixes;
  failure, the immediate failure state of s.

Procedure Initialize()
  let $root = newstate$
  $root.scheme\_prefixes = \emptyset$, $root.subscheme\_prefixes = \emptyset$
  for $t \in Red'_\Sigma$
    let $p = newprefix$, $p.terms[0] = t$, $p.distinguished = \{0\}$
    if t is a functional subscheme
      $root.subscheme\_prefixes = root.subscheme\_prefixes \cup \{p\}$
    else $root.scheme\_prefixes = root.scheme\_prefixes \cup \{p\}$
  end for
  $root.indexes = \{0\}$, $root.index = 0$, $root.successors = \emptyset$,
  $root.psize = 1$ $root.failure =$ **undefined**
end Initialize

Procedure Advance(s)
  while $s.scheme\_prefixes \neq \emptyset$ **and** $s.subscheme\_prefixes \neq \emptyset$ **do**
    if $s.scheme\_prefixes \neq \emptyset$ **select any** $p \in s.scheme\_prefixes$
    else select any $p \in s.subscheme\_prefixes$
    end if

```
        s.scheme_prefixes = s.scheme_prefixes \ {p} //(1)
        let t = p.terms[s.index]
        let f = t.symbol
        for i ∈ [0..arity(f) − 1] do
            p.terms[s.psize + i] = t.child[i]
            if t.child[i] ≠ Ω
                p.distinguished = p.distinguished ∪ {s.psize + i}
            end if
        end do
        p.distinguished = p.distinguished \ {s.index}
        if ∃s' such that < f, s' >∈ s.successors
            let s' = a state | < f, s' >∈ s.successors
            if Φ(s') = root // (2)
                s'.indexes = s'.indexes ∩ p.distinguished// (3)
            end if
            if p is a scheme prefixe
                s'.scheme_prefixes = s'.scheme_prefixes ∪ {p}
            else s'.subscheme_prefixes = s'.subscheme_prefixes ∪ {p}
            end if
        elseif p is a scheme prefixe
        // we don't create a new state for a subscheme
            let s' = newstate
            s'.successors = ∅, s'.scheme_prefixes = {p},
            s'.psize = s.psize + f.arity
            let t = s.failure
            while t ≠ root∧ ∄ < f, u >∈ t.successors do
                t = t.failure
            if ∃ < f, u >∈ t.successors
                s'.failure = u, s'.index = u.index + s'.psize − u.psize
            else
                s'.failure = root, s'.index = undefined,
                s'.indexes = p.distinguished
            end if
            s.successors = s.successors ∪ {< f, s' >}
        end if
    end while
    for < f, s' >∈ s.successors do
        if s'.index =undefined
            if s'.indexes = ∅ and |s'.subscheme_prefixes| ≠ 0
               and |s'.scheme_prefixes| ≠ 1
                error "not forward branching"
            else s'.index = any i ∈ s'.indexes
            end if
        end if
    end for
end Advance
```

5 Explanation of the Algorithm

The main trick of the algorithm lies in the encoding of a scheme (or subscheme) prefix. An array is used to index the known parts of the prefix. The array is constructed dynamically according to the order in which a pattern or a subpattern is scanned.

The algorithm constructs the index tree top-down in a breadth-first manner. That way, we are sure that a failure node exists when needed. Each state of the index tree keeps a set of scheme prefixes and a set of subscheme prefixes. The schemes prefixes are prefixes of ordinary left-hand side schemes. The subscheme prefixes are prefixes of subschemes for which the root symbol is a functional symbol.

To initalize the algorithm, we create the initial state of the index tree. Then the two prefix sets are initialized with all of the empty prefixes of the schemes and the functional subschemes respectively. We know that the index point to use for the initial state is Λ, which corresponds to index 0 of the term array in the prefixes.

Following the initialization, we use a worklist organized in the form of a queue in order to partition the prefixes of a state into subsets, each corresponding to a successor state. This partitioning is done by the `Advance` procedure.

The `Advance` procedure works in two steps. The first step advances all the scheme prefixes and the second step advances all the functional subscheme prefixes; the order is important. Advancing a scheme for a given index i consists of refining the scheme at i and then looking for a state corresponding to the symbol at i. If such a state does not exist *and* the scheme is not a functional subscheme, then it must be created as described below. We don't want to create a state for a functional subscheme since we are not interested in matching them; their main purpose is overlap detection. Also note that, since schemes are processed before subschemes, there is no risk of throwing away a subscheme overlapping with a scheme.

When the state is found, the algorithm checks whether its set of indexes needs to be updated according to property 1; if so, a set intersection is performed. To determine whether the update must be done, the algorithm can check whether the index has already been chosen or, equivalently, whether the associated failure state is *root*. Note that the later solution depends on the state, not on the scheme. Thus, for a given state, the result of the test is fixed and we could save some tests by code duplication. This is due to the fact that if the algorithm cannot choose the index at state creation, as explained in the next paragraph, it cannot choose an index in the set of possible *indexes* before having processed all the schemes and subschemes associated with that state.

When we create a state, we first compute its failure transition. For that, we can use the *failure transition lemma* [Dur94] and Definition 6 to apply the following technique: to find the failure transition for a state s knowing the failure transition chain issued from its predecessor p, we just need to find the first failure point t of this chain such that $\delta(t, F) = r$ is defined. F is the symbol satisfying $\delta(p, F) = s$. If such a state does not exist, then $\Phi(s) = root$. Note that this operation is performed in $\mathcal{O}(j)$ where j is the distance between the state s and the root. If a non-root failure state is found, we can use the choice procedure described in case 2 of the automaton construction on page 6 as the index of the new state, which corresponds to a simple arithmetic operation.

6 Complexity

6.1 Time

We now consider the complexity of the algorithm in terms of set intersection operations. The reader can check that it corresponds to the most expensive set of operations by looking at the explanations of Section 5, page 9, and by considering the fact that, given a state s and a scheme p, p is considered only once. See instruction (1) of the algorithm.

The global strategy is to perform a reassociation of cost in order to show that the work performed on the subschemes can be used to avoid similar work on schemes. But first, we need some definitions.

Notation 2 *A state s is said to be* stable *iff $\Phi(s) = root$.*

Definition 7 *Given an index tree I, we define its* considered data set *cd_I as the set of pairs (s,p) where s is a state of I and p a scheme or subscheme associated with s (i.e., considered by the algorithm while processing state s; see the tagged instructions (1)(2)(3)). We drop I when it is implied by the context. s and p correspond to the variables of the same names in the algorithm. For p, note that there is a bijection betwen the prefixes and the schemes.*

Next, we define a set of projections of cd:

Notation 3 *We call cd_{stb} the restriction of cd to stable states and cd_{nstb} its complementary set.*

We call cd^{sc} the restriction of cd to schemes and cd^{ssc} its restriction to proper subschemes.

We can combine these two types of notation in a direct way. For example, cd^{ssc}_{nstb} denotes the restriction of cd to non-stable states and subschemes.

Definition 8 *We define the elementary potential cost of (s,p), noted $epc(s,p)$ as the cost of the set intersection (3_x) performed for p at state s, supposing that the test (2_x) is not performed (or systematically returns true).*

We define the elementary real cost of (s,p), noted $erc(s,p)$ as the cost of the set operation actually generated by p at state s:

$$erc(s,p) = \begin{cases} epc(s,p) & \text{if test } (2_x) \text{ produces } \texttt{true} \\ 0 & \text{otherwise} \end{cases}$$

We call rc the global real cost in term of set intersection operations: $rc = \sum_{(s,p)\in cd} erc(s,p)$

The essence of our proof consists of showing that $rc = \sum_{(s,p)\in cd_{stb}} erc(s,p) \leq \sum_{(s,p)\in cd^{sc}} epc(s,p)$. And the first step of the proof can be stated like this:

Lemma 1 $rc = \sum_{(s,p)\in cd_{stb}} epc(s,p)$

Proof. The algorithm performs set intersection operations only for state s such that $\Phi(s) = root$; otherwise, the computation of failure points and indexes can be done directly. See Section 5. So, we have $erc(s,p) = \begin{cases} epc(s,p) & \text{if } s \in cd_{stb} \\ 0 & \text{otherwise} \end{cases} \Rightarrow$ $\sum_{(s,p)\in cd} erc(s,p) = \sum_{(s,p)\in cd_{stb}} epc(s,p)$ •

For the next step, we need to define the function *trnsf* that we will use to make the reassociation of cost.

Definition 9 *We define* $trnsf : cd^{ssc} \mapsto cd^{sc}_{nstb}$ *as* $trnsf((M/u, w/u), p/u) = ((M, w), p)$ *where* p *is a scheme. The function trnsf is well defined thanks to the construction lemma [Dur94] and Definition 4.*

Intuitively, the function *trnsf* maps the inspection of an occurrence of a subscheme to the inspection of the corresponding occurrence of the scheme from which the subscheme is extracted. Note that such a mapping traverses a chain of failure transitions backwards (according to Definition 4), as shown in Figure 1. The reassociation of cost can be performed thanks to the following lemma:

Lemma 2 $trnsf_{|cd^{ssc}_{stb}} \mapsto cd^{sc}_{nstb}$ *is bijective.*

Proof. $trnsf_{|cd^{ssc}_{stb}}$ is clearly surjective since it traverses a chain of failure transitions backwards. Injectiveness comes from the fact that if $trnsf_{|cd^{ssc}_{stb}}(s_1, p_1) = trnsf_{|cd^{ssc}_{stb}}(s_2, p_2) = (s, p)$, s_1 and s_2 are both failure points of s, so, one must be a failure point for the other; but that idea contradicts the hypothesis that they are both stable. •

Lemma 3 $rc \leq \sum_{(s,p)\in cd^{sc}} epc(s, p)$.

Proof. We have (Lemma 1): $rc = \sum_{(s,p)\in cd_{stb}} epc(s, p) = \sum_{(s,p)\in cd^{sc}_{stb}} epc(s, p) + \sum_{(s,p)\in cd^{ssc}_{stb}} epc(s, p)$.

Since set intersection is linear in the size of the sets, we have $epc(s, p) \leq epc(trnsf(s, p))$; thus $rc \leq \sum_{(s,p)\in cd^{sc}_{stb}} epc(s, p) + \sum_{(s,p)\in cd^{ssc}_{stb}} epc(trnsf(s, p))$. Then, (Lemma 2) $rc \leq \sum_{(s,p)\in cd^{sc}_{stb}} epc(s, p) + \sum_{(s,p)\in cd^{sc}_{nstb}} epc(s, p)$. That concludes the proof of the lemma. •

Theorem 2 $rc \leq \sum_{L_i\in Red} size(L_i)^2$

Proof. Thanks to Lemma 3, we can say that $rc \leq \sum_{L_i\in Red} \sum_{(s,L_i)\in cd} epc(s, L_i)$. Since $|\{(s, L_i) \in cd\}| = size(L_i)$ (the entire term is inspected) and $epc(s, L_i) \leq size(L_i)$, we have $rc \leq \sum_{L_i\in Red} size(L_i)^2$ •

6.2 Space

We need to allocate memory for the states and for the prefixes. The memory needed the states is in $\mathcal{O}(\sum_{L_i\in Red} size(L_i))$. The memory needed for the prefixes, for this algorithm, is in $\mathcal{O}(\sum_{L_i\in Red'} size(L_i))$ But it's straightforward - using sharing - to implement the prefixes in order to get $O(\sum_i |n_i|)$.

This is an important improvement, compared to [Dur94], since the previous algorithm had to manage a memory space in $\mathcal{O}((\sum_{L_i\in Red} size(L_i))^2)$.

7 Conclusion

We have presented a new algorithm for recognizing the forward-branching class of term rewriting systems. Our algorithm has a time and space complexity of $\mathcal{O}(\sum_{L_i\in Red} size(L_i)^2)$, which is important since our programs are typically composed of many small left-hand sides.

Moreover, both the algorithm and the complexity proof are simplier. Note that an implementation exist, that will soon be integrated in the distributed version of our system.

References

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), 1975.

[DS90] Irène Durand and Robert Strandh. A decision procedure for forward-branching equational programs. Technical Report 05-90, GRECO Programmation, 1990.

[DS91] Irène Durand and Robert Strandh. A simple characterization of the class of forward-branching equational programs. Technical Report 04–91, GRECO Programmation, 1991.

[Dur94] Irène Durand. Bounded, strongly sequential and forward-branching term rewriting systems. *Journal of Symbolic Computation*, 18:319–352, 1994. Also Technical Report LaBRI 92–23.

[Dur95] Irène Durand. Deciding strong sequentiality for orthogonal term rewriting systems is in co-np. Technical Report 1090-95, LaBRI, 1995.

[HL79] Gérard Huet and Jean-Jacques Lévy. Computations in non-ambiguous linear term rewriting systems. Technical Report 359, INRIA, 1979.

[HO82] Christopher Hoffmann and Michael J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, pages 83–112, 1982.

[KM91] Jan Willem Klop and Aart Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

[KP78] G. Kahn and G. Plotkin. Domaines concrets. Technical Report 336, INRIA, 1978.

[O'D85] Michael J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[Ros73] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

[SR90] R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *Proceedings of Fifth IEEE Symposium on Logic in Computer Science*, pages 230–241, 1990.

[Str88] Robert I. Strandh. *Compiling Equational Programs into Efficient Machine Code*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1988.

[Str89] Robert I. Strandh. Classes of equational programs that compile into efficient machine code. In *Proceedings of the Third International Conference on Rewrite Techniques and Applications*, 1989.