Lazy AC-Pattern Matching for Rewriting

Walid Belkhir and Alain Giorgetti
FEMTO-ST, University of Franche-Comté,
16 route de Gray, 25030 Besançon cedex, France
INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy, France
{walid.belkhir,alain.giorgetti}@femto-st.fr

We define a lazy pattern-matching mechanism modulo associativity and commutativity. The solutions of a pattern-matching problem are stored in a lazy list composed of a first substitution at the head and a non-evaluated object that encodes the remaining computations. We integrate the lazy AC-matching in a strategy language: rewriting rule and strategy application produce a lazy list of terms.

1 Introduction

Term rewriting modulo associativity and commutativity of some function symbols, known as AC-rewriting, is a key operation in many programming languages, theorem provers and computer algebra systems. Examples of AC-symbols are $+$ and $*$ for addition and multiplication in arithmetical expressions, $\vee$ and $\wedge$ for disjunction and conjunction in Boolean expressions, etc. AC-rewriting performance mainly relies on that of its AC-matching algorithm. On the one hand, the problem of deciding whether an AC-matching problem has a solution is NP-complete [2]. On the other hand, the number of solutions to a given AC-matching problem can be exponential in the size of its pattern. Thus many works propose optimizations for AC-matching. One can divide optimized algorithms in two classes, depending on what they are designed for. In the first class some structural properties are imposed on the terms, and the pattern falls into one of several forms for which efficient algorithms can be designed. Examples are the depth-bounded patterns in the many-to-one matching algorithm used by Elan [12] and greedy matching techniques adopted in Maude [6]. In the second class there is no restriction on the structural properties of the terms. Algorithms in this class are search-based, and use several techniques to collapse the search space, such as constraint propagation on non linear variables [11], recursive decomposition via bipartite graphs [8], ordering matching subproblems based on constraint propagation [9] and Diophantine techniques [10].

Formal semantics proposed so far for AC-rewriting enumerate all the possible solutions of each matching problem. More precisely, the application modulo AC of a rewrite rule $l \to r$ to a given term t usually proceeds in two steps. Firstly, all the solutions (i.e. substitutions) σ_1, ..., σ_n ($n \geq 0$) of the AC-matching problem whether the term t matches the pattern l are computed and stored in a structure, say a set $\{\sigma_1, \ldots, \sigma_n\}$. Secondly, this set is applied to r and the result is the set $\{\sigma_1(r), \ldots, \sigma_n(r)\}$. Other structures such as multisets or lists can alternatively be used for other applications of the calculus. Directly implementing this *eager* semantics is clearly less efficient than a lazy mechanism that only computes a first result of the application of a rewrite rule and allows the computation by need of the remaining results. As far as we know no work defines the AC-matching in a lazy way and integrates it in a rewriting semantics.

Another motivation of this work lies in our involvement in the formulation of the homogenization of partial derivative equations within a symbolic computation tool [16, 17]. For this purpose, we have

S. Escobar (Ed.): Workshop on Reduction Strategies
in Rewriting and Programming (WRS2011)
EPTCS 82, 2012, pp. 37–51, doi:10.4204/EPTCS.82.3

designed and developed a rule-based language called `symbtrans` [1] for "symbolic transformations" built on the computer algebra system Maple. Maple pattern-matching procedures are not efficient and its rewriting kernel is very elementary. Besides, Maple is a strict language, it does not provide any laziness feature. We plan to extend `symbtrans` with AC-matching.

In this paper we first specify a lazy AC-matching algorithm which computes the solutions of an AC-matching problem by need. Then we integrate the lazy AC-matching in a strategy language. In other words we define a lazy semantics for rule and strategy functional application. Our goal is to specify the lazy AC-matching and strategy semantics towards an implementation in a strict language, such as Maple. We reach this goal by representing lazy lists by means of explicit objects.

The paper is organized as follows. Section 2 introduces some terminology and notations. Section 3 shows a connection between AC-matching and surjective functions, used in the remainder of the paper. Section 4 formally defines a lazy semantics of AC-matching (with rewrite rules). It states its main properties and shows how it differs from an eager semantics. Section 5 integrates the lazy AC-matching in the operational semantics of a rule application on a term, first at top position, and then at other positions, through classical traversal strategies. Section 6 presents a prototypal implementation of lazy AC-matching and some experimental results derived from it. Section 7 concludes.

2 Notation and preliminaries

Let $[n]$ denote the finite set of positive integers $\{1,\ldots,n\}$ and let $|S|$ denote the cardinality of a finite set S. Thus, in particular, $|[n]| = n$.

Familiarity with the usual first-order notions of signature, (ground) term, arity, position and substitution is assumed. Let $\mathscr{X}$ be a countable set of variables, $\mathscr{F}$ a countable set of function symbols, and $\mathscr{F}_{AC} \subseteq \mathscr{F}$ the set of associative-commutative function symbols. Let $\mathcal{T}$ denote the set of terms built out of the symbols in $\mathscr{F}$ and the variables in $\mathscr{X}$. Let $\mathcal{S}$ denote the set of substitutions $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ with variables $x_1,\ldots,x_n$ in $\mathscr{X}$ and terms $t_1,\ldots,t_n$ in $\mathcal{T}$. If t is a term and σ is a substitution then $\sigma(t)$ denotes the term that results from the application of σ to t. Given a position p, the subterm of t at position p is denoted by $t_{|p}$. We shall write t_ε for the symbol at the root of term t, i.e. $t = t_\varepsilon(t_1,\ldots,t_n)$.

A term t in $\mathcal{T}$ is *flat* if, for any position p in t, $t_{|p} = +(t_1,\ldots,t_n)$ for some symbol $+$ in $\mathscr{F}_{AC}$ implies that the root symbol $(t_i)_\varepsilon$ of each direct subterm t_i $(1 \le i \le n)$ is not $+$. We denote by $\sharp_{AC}(t)$ the number of AC-symbols in the term t.

$\mathbb{T}$-Matching. For an equational theory $\mathbb{T}$ and any two terms t and t' in $\mathcal{T}$ we say that t matches t' modulo $\mathbb{T}$ and write $t \ll_{\mathbb{T}} t'$ iff there exists a substitution σ in $\mathcal{S}$ s.t. $\mathbb{T} \models (\sigma(t) = t')$. In this paper the theory $\mathbb{T}$ is fixed. It is denoted *AC* and axiomatizes the associativity and commutativity of symbols in $\mathscr{F}_{AC}$, i.e. it is the union of the sets of axioms $\{t_1 + t_2 = t_2 + t_1,\ (t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)\}$ when $+$ ranges over $\mathscr{F}_{AC}$.

Rule-based semantics The following sections define the semantics of AC-matching, rule application and strategy application by rewriting systems composed of labeled rewriting rules of the form `rule_label:` $\cdot \rightsquigarrow \cdot$, where the rewrite relation $\rightsquigarrow$ should not be confused with the relation $\rightarrow$ of the rewriting language. This semantics is said to be "rule-based".

3 AC-matching and surjections

Let $+ \in \mathscr{F}_{AC}$ be some associative and commutative function symbol. This section first relates a restriction of the pattern-matching problem $+(t_1,\ldots,t_k) \ll_{AC} +(u_1,\ldots,u_n)$, where $1 \leq k \leq n$, with the set $S_{n,k}$ of surjective functions (*surjections*, for short) from $[n]$ to $[k]$. Then a notation is provided to replace surjections with their rank to simplify the subsequent exposition.

Definition 3.1 (Application of a surjection on a term) Let $n \geq k \geq 1$ be two positive integers, $u = +(u_1,\ldots,u_n)$ be a term and $s \in S_{n,k}$ be a surjection from $[n]$ to $[k]$. The *application of s on u* is defined by $s(u) = +(\alpha_1,\ldots,\alpha_k)$ where $\alpha_i = u_j$ if $s^{-1}(\{i\}) = \{j\}$ and $\alpha_i = +(u_{j_1},\ldots,u_{j_m})$ if $s^{-1}(\{i\}) = \{j_1,\ldots,j_m\}$ and $j_1 < \ldots < j_m$ with $m \geq 2$.

Example 3.1 The application of the surjection $s = \{1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 1\}$ on the term $+(u_1, u_2, u_3, u_4)$ is $s(+(u_1,u_2,u_3,u_4)) = +(u_4,+(u_1,u_2),u_3)$.

The following proposition, whose proof is omitted, relates a subclass of AC-matching problems with a set of surjections.

Proposition 3.1 *Let t and u be two flat terms with the same AC-symbol at the root and containing no other AC-symbol than the one at the root. Let k (resp. n) be the arity of the root of t (resp. u). Then the matching problem $t \ll_{AC} u$ and the conjunction of matching problems $\bigwedge_{s \in S_{n,k}} t \ll_{\emptyset} s(u)$ admit the same set of solutions.*

For $n \geq k \geq 1$, an integer $rk(s)$ in $\{1,\ldots,|S_{n,k}|\}$ can be associated one-to-one to each surjection s in $S_{n,k}$. It is called the *rank* of s. It will be used in the subsequent sections to iterate over the set $S_{n,k}$. For each term u of arity $n \geq 1$, each $1 \leq k \leq n$, and each integer i in $\{1,\ldots,|S_{n,k}|\}$, let $unrk(i)$ denote the surjection with rank i. The application of the integer i to the term u, denoted by $i(u)$, is defined by $i(u) = unrk(i)(u)$, where application of a surjection to a term is defined by Def. 3.1.

4 AC-matching

This section defines an eager and a lazy semantics for pattern-matching modulo associativity and commutativity. By "eager" we mean a rewriting system specifying the computation of all the solutions to a given matching problem, without any mechanism to delay the production of other solutions when a first one is produced. The AC-matching execution steps are made explicit by introduction of a syntactic representation of matching problems by matching constraints. They are a generalization to AC-matching of the matching constraints defined e.g. in [4] for syntactic matching (i.e. for the empty theory). Both semantics of AC-matching are given in terms of a conditional rewrite system on these constraints. In what follows all the terms are assumed to be flat.

4.1 Eager AC-matching

Figure 1 proposes a rule-based *eager* semantics for AC-matching. This system (named *Eager*) reduces *eager matching constraints* inductively defined by the grammar $\mathcal{E} ::= \mathbf{F} \mid \mathbf{I} \mid \mathcal{T} \ll_{AC} \mathcal{T} \mid \mathcal{E} \wedge \mathcal{E} \mid \mathcal{E} \vee \mathcal{E}$. This definition and the notations for matching constraints adapt and extend the ones from the ρ-calculus with explicit matching [4]. The constraint $\mathbf{F}$ denotes an absence of solution (failure). The constraint $\mathbf{I}$ denotes the identity substitution resulting from an initial trivial matching problem. The expression

$p \ll_{AC} t$ denotes the elementary matching problem whether the term t matches the pattern p modulo AC. The symbol $\wedge$ is the constructor of conjunctions of constraints. The symbol $\vee$ is introduced to enumerate various solutions as lists of constraints. Its priority is lower than that of $\wedge$. Both are assumed associative. Then the notation $C_1 \wedge \ldots \wedge C_n$ is used without ambiguity, and similarly for $\vee$, for $n \geq 1$. The constraint $\mathbf{I}$ is a neutral element for $\wedge$. The constraint $\mathbf{F}$ is a neutral element for $\vee$ and an absorbing element for $\wedge$.

`E_match_AC:`	$\underbrace{+(t_1,\ldots,t_k)}_{t} \ll_{AC} \underbrace{+(u_1,\ldots,u_n)}_{u} \rightsquigarrow \bigvee_{j=1}^{j=\lvert S_{n,k}\rvert} \bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i$ if $+ \in \mathscr{F}_{AC}$, $k \leq n$ and $j(u) = +(\alpha_1,\ldots,\alpha_k)$	
`E_match:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} t_\varepsilon(u_1,\ldots,u_n) \rightsquigarrow \bigwedge_{i=1}^{i=n} t_i \ll_{AC} u_i$	if $t_\varepsilon \in \mathscr{F} \setminus \mathscr{F}_{AC}$
`E_match_AC_fail:`	$+(t_1,\ldots,t_k) \ll_{AC} +(u_1,\ldots,u_n) \rightsquigarrow \mathbf{F}$	if $+ \in \mathscr{F}_{AC}$ and $k > n$
`E_match_fail:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} u_\varepsilon(u_1,\ldots,u_m) \rightsquigarrow \mathbf{F}$	if $t_\varepsilon \neq u_\varepsilon$
`E_fail_gen:`	$x \ll_{AC} t \wedge E \wedge x \ll_{AC} t' \rightsquigarrow \mathbf{F}$	if $x \in \mathscr{X}$ and $t \neq t'$
`E_var_clash:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} x \rightsquigarrow \mathbf{F}$	if $x \in \mathscr{X}$
`E_DNF_1:`	$F \wedge (G \vee H) \rightsquigarrow (F \wedge G) \vee (F \wedge H)$	
`E_DNF_2:`	$(G \vee H) \wedge F \rightsquigarrow (G \wedge F) \vee (H \wedge F)$	

Figure 1: *Eager* system of AC-matching rules

In Figure 1, the notation $\bigwedge_{i=1}^{i=n} t_i \ll_{AC} u_i$ stands for $\mathbf{I}$ if $n = 0$ and for $t_1 \ll_{AC} u_1 \wedge \ldots \wedge t_n \ll_{AC} u_n$ otherwise. The rule `E_match_AC` corresponds to Proposition 3.1 when t_1, …, t_k, u_1, …, u_n contain no AC symbol, and generalizes it otherwise. The positive integer j iterates over surjection ranks. The rules `E_match`, `E_match_fail`, `E_fail_gen`, and `E_var_clash` are the same as in syntactic pattern-matching. With AC symbols, they are completed with the rule `E_match_AC_fail`. The rules `E_DNF_1` and `E_DNF_2` correspond to the normalization of constraints into a disjunctive normal form, DNF for short. A constraint is in DNF if it is of the form $\vee_i \wedge_j F_{i,j}$, where $F_{i,j}$ is a constraint not containing $\vee$ and $\wedge$.

It is standard to show that the system *Eager* is terminating. The rules `E_DNF_1` and `E_DNF_2` make it not confluent, but the following post-processing reduces to a unique normal form all the irreducible constraints it produces from a given pattern-matching problem. The post-processing consists of *(i)* replacing the trivial constraints of the form $x \ll_{AC} x$ by $\mathbf{I}$, *(ii)* replacing each non-trivial constraint $x \ll_{AC} t$ by the elementary substitution $x \mapsto t$, *(iii)* eliminating duplicated elementary substitutions by replacing each expression of the form $\wedge_{i=1}^{i=n} E_i$ (with $n \geq 1$) by the set $\bigcup_{i=1}^{i=n} \{E_i\}$ that represents a non-trivial substitution, then *(iv)* replacing $\mathbf{I}$ by $\{\}$ and finally *(v)* replacing each disjunction of substitutions $\vee_{j=1}^{j=n} S_j$ with $n \geq 1$ by the set $\bigcup_{j=1}^{j=n} \{S_j\}$, that represents a set of substitutions. A constraint in normal form can be either $\mathbf{F}$, if there is no solution to the initial matching problem, or a non-empty set $\{\sigma_1,\ldots,\sigma_n\}$ of substitutions which corresponds to the set of all solutions of the matching problem. In particular σ_i may be $\{\}$, for some $i \in [n]$.

The theory for the associative symbols $\wedge$ and $\vee$ deliberately excludes commutativity, because they appear in the pattern of some rules of the *Eager* system. We now motivate this design choice. Since

the *Eager* system is terminating and confluent – in the sense explained above – we can consider it as an algorithm and implement it without modification, in a programming language supporting pattern-matching modulo the theory of constraint constructors. Thus it would be ill-founded to require that the language supports AC-matching, in order to extend it precisely with an AC-matching algorithm! In this aspect our approach differs from the one of [5], whose more compact calculus handles "result sets", i.e. the underlying theory includes associativity, commutativity and idempotency, but matching with patterns containing set constructors is implemented by explorations of set data structures.

4.2 Lazy AC-matching

We now define a lazy semantics for pattern-matching modulo associativity and commutativity, as a rewriting system named *Lazy*. It reduces constraints defined as follows.

Definition 4.1 The set of *delayed matching constraints*, hereafter called *constraints* for short, is inductively defined by the grammar: $\mathcal{C} ::= \mathbf{F} \mid \mathbf{I} \mid \mathcal{T} \ll_{AC} \mathcal{T} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \mid \mathit{Next}(\mathcal{C}) \mid \langle \mathcal{T}, \mathcal{T}, \mathbb{N}^\star \rangle$.

The first five constructions have the same meaning as in eager matching constraints. As in the eager case, the symbols $\wedge$ and $\vee$ are associative, the constraint $\mathbf{F}$ is an absorbing element for $\wedge$, and the constraint $\mathbf{I}$ is a neutral element for $\wedge$. However, the constraint $\mathbf{F}$ is no longer a neutral element for $\vee$. The construction $\mathit{Next}(C)$ serves to activate the delayed computations present in the constraint C. When the terms t and u have the same AC symbol at the root, the constraint $\langle t, u, s \rangle$ denotes the delayed matching computations of the problem $t \ll_{AC} u$ starting with the surjection with rank s, and hence, the matching computations for all the surjections with a rank s' s.t. $s' < s$ have already been performed. The conditions that t and u have the same AC symbol at the root and that the arities of t and u correspond to the domain and codomain of the surjection with rank s are not made explicit in the grammar, but it would be easy to check that they always hold by inspecting the rules of the forthcoming system *Lazy*. Delayed matching constraints of the form $\langle t, u, s \rangle$ and satisfying these conditions are more simply called *triplets*.

Figure 2 defines the first part of the lazy semantics, a rewriting system named $\mathscr{R}_1$. The rules `match_AC_fail`, `match`, `match_fail`, `var_clash` and `fail_gen` are standard and already appeared in the eager matching system. The rule `match_AC` activates the delayed matching computations starting from the first surjection. It is immediately followed by the rule `match_surj_next` from the rewriting system $\mathscr{R}_2$ defined in Figure 3.

`match_AC:`	$\underbrace{+(t_1,\ldots,t_k)}_{t} \ll_{AC} \underbrace{+(u_1,\ldots,u_n)}_{u} \rightsquigarrow \mathit{Next}(\langle t, u, 1 \rangle)$	if $+ \in \mathscr{F}_{AC}$ and $k \leq n$
`match:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} t_\varepsilon(u_1,\ldots,u_n) \rightsquigarrow \bigwedge_{i=1}^{i=n} t_i \ll_{AC} u_i$	if $t_\varepsilon \in \mathscr{F} \setminus \mathscr{F}_{AC}$
`match_AC_fail:`	$+(t_1,\ldots,t_k) \ll_{AC} +(u_1,\ldots,u_n) \rightsquigarrow \mathbf{F}$	if $+ \in \mathscr{F}_{AC}$ and $k > n$
`match_fail:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} u_\varepsilon(u_1,\ldots,u_m) \rightsquigarrow \mathbf{F}$	if $t_\varepsilon \neq u_\varepsilon$
`fail_gen:`	$x \ll_{AC} t \wedge C \wedge x \ll_{AC} t' \rightsquigarrow \mathbf{F}$	if $x \in \mathscr{X}$ and $t \neq t'$
`var_clash:`	$t_\varepsilon(t_1,\ldots,t_n) \ll_{AC} x \rightsquigarrow \mathbf{F}$	if $x \in \mathscr{X}$

Figure 2: $\mathscr{R}_1$ system: AC-matching rules

`fail_next:`	$\mathbf{F} \vee C \rightsquigarrow \mathit{Next}(C)$	`next_fail:`	$\mathit{Next}(\mathbf{F}) \rightsquigarrow \mathbf{F}$
`next_id:`	$\mathit{Next}(\mathbf{I}) \rightsquigarrow \mathbf{I}$	`next_basic:`	$\mathit{Next}(x \ll_{AC} u) \rightsquigarrow x \ll_{AC} u$
`next_and:`	$\mathit{Next}(C_1 \wedge C_2) \rightsquigarrow \mathit{Next}(C_1) \wedge \mathit{Next}(C_2)$		
`next_or:`	$\mathit{Next}(C_1 \vee C_2) \rightsquigarrow \mathit{Next}(C_1) \vee C_2$, if $C_1 \neq \mathbf{F}$		
`match_surj_next:`	$\mathit{Next}(\langle t,u,s\rangle) \rightsquigarrow (\bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i) \vee \langle t,u,s+1\rangle$ if $t = t_\varepsilon(t_1,\ldots,t_k)$, $u = t_\varepsilon(u_1,\ldots,u_n)$, $s < \lvert S_{n,k}\rvert$ and $s(u) = t_\varepsilon(\alpha_1,\ldots,\alpha_k)$		
`match_surj_last:`	$\mathit{Next}(\langle t,u,\lvert S_{n,k}\rvert\rangle) \rightsquigarrow \bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i$ if $t = t_\varepsilon(t_1,\ldots,t_k)$, $u = t_\varepsilon(u_1,\ldots,u_n)$ and $\lvert S_{n,k}\rvert(u) = t_\varepsilon(\alpha_1,\ldots,\alpha_k)$		

Figure 3: $\mathscr{R}_2$ system: *Next* rules

In $\mathscr{R}_2$ the rule `fail_next` states that the presence of a failure activates the delayed computations in the constraint C. The other rules propagate the activation of the delayed computations on the inductive structure of constraints. The rule `next_and` propagates the *Next* constructor to sub-constraints. The rule `next_or` propagates the *Next* constructor to the head of a list of constraints, provided this head is not $\mathbf{F}$.

When the constraint is $\langle t,u,s\rangle$, two cases have to be considered. If the surjection rank $s < |S_{n,k}|$ is not the maximal one, then the rule `match_surj_next` reduces the constraint $\mathit{Next}(\langle t,u,s\rangle)$ to a set of matching constraints according to the surjection with rank s, followed by delayed computations that will be activated from the next surjection, with rank $s+1$. In the final case when $s = |S_{n,k}|$ (rule `match_surj_last`), there is no delayed computations.

`DNF_1:`	$F \wedge (G \vee H) \rightsquigarrow (F \wedge G) \vee (F \wedge H)$
`DNF_2:`	$(G \vee H) \wedge F \rightsquigarrow (G \wedge F) \vee (H \wedge F)$

Figure 4: $\mathscr{R}_3$ system : DNF rules

Rules for the reduction of constraints in DNF are the same as in the eager system, but are defined in the separate rewrite system $\mathscr{R}_3$ given in Figure 4.

Let us denote by *Lazy* the rewriting system $\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3$ composed of the rules of Figures 2, 3 and 4, equipped with the evaluation strategy *sat* that consists of the iteration of the following process: *(i)* Applying the rules of $\mathscr{R}_1 \cup \mathscr{R}_2$ until no rule is applicable, then *(ii)* applying the rules of $\mathscr{R}_3$ until no rule is applicable.

4.3 Termination of lazy AC-matching

We prove the termination of the lazy AC-matching system *Lazy* modularly, by considering some of its sub-systems separately, in the following lemmas.

Lemma 4.1 *The rewriting system $\mathscr{R}_2$ is terminating.*

Proof. On the one hand, no rule in $\mathscr{R}_2$ produces the $\mathbf{F}$ constraint. Thus, after replacing with $\mathit{Next}(C)$ the occurrences of $\mathbf{F} \vee C$ in the input constraint, the rule `fail_next` is no longer used. On the other hand, no rule in $\mathscr{R}_2$ can reduce the right side of the rules `match_surj_next` and `match_surj_last`. Thus it is sufficient to prove the termination of $\mathscr{R}_4 = \{$`next_fail`, `next_id`, `next_basic`, `next_and`, `next_or`$\}$.

This is standard, using a recursive path ordering [7]. Intuitively, the termination of $\mathscr{R}_4$ is ensured by the fact that all the rules in $\mathscr{R}_4$ push the *Next* constructor down until reaching the leaves. □

Before proving the termination of $\mathscr{R}_1 \cup \mathscr{R}_2$, we need to introduce a variant of terms and triplets to prove a technical lemma about their occurrences in derivations. The *marked term* t^M associated to the term t is obtained from t by replacing each AC symbol $+$ at some position p in t with $+_p$. The *marked triplet* associated to the triplet $\langle t,u,s\rangle$ is defined to be $\langle t^M,u,s\rangle$. The *marked constraint* associated to the constraint $t \ll_{AC} u$ is $t^M \ll_{AC} u$. The *marked variant* $\mathscr{R}_1^M$ (resp. $\mathscr{R}_2^M$, $Lazy^M$) of the $\mathscr{R}_1$ (resp. $\mathscr{R}_2$, *Lazy*) system is obtained from the latter by replacing triplets with marked triplets, constraints with marked constraints and $+(t_1,\ldots,t_k)$ by $+_p(t_1,\ldots,t_k)$ for any p, in the pattern of the rules `match_AC` and `match_AC_fail`. It is clear, and thus admitted, that the derivations of $Lazy^M$ are the variants of the derivations of *Lazy*, in a natural sense.

Lemma 4.2 *Let t^M be a marked term with an AC symbol at the root and u be a term with the same AC symbol at the root. Consider a derivation $t^M \ll_{AC} u \rightsquigarrow C_1 \rightsquigarrow C_2 \rightsquigarrow \ldots$ with rules in $\mathscr{R}_1^M \cup \mathscr{R}_2^M$. Then, the number of marked triplets in the sequence $C_1, C_2, \ldots$ is strongly bounded, in the following sense:* (i) *There is an upper bound for the number of marked triplets in each C_i and* (ii) *for each subterm t' of t with an AC symbol at the root, if a marked triplet built up on the marked subterm t' (i.e. of the form $\langle t',u',s\rangle$ for some u' and s) is deleted from some $C_i, i \geq 1$, then it never appears again in C_j, for all $j > i$.*

Proof. (i) The variant of the rule `match_AC` replaces an elementary matching problem (between two terms with an AC symbol at the root) with a marked triplet. It is the only rule of $\mathscr{R}_1^M \cup \mathscr{R}_2^M$ that produces a marked triplet whose first (marked) term is new. Then the variant of the rule `match_surj_next` replaces a marked triplet whose first (marked) term is the subterm $t' = +_p(\ldots)$ of t at some position p with a marked triplet on the same terms, whose surjection rank is incremented. Thus the number of marked triplets in a constraint is bounded above by the number of positions of AC symbols in t.

(ii) The only rule of $\mathscr{R}_1^M \cup \mathscr{R}_2^M$ that deletes a marked triplet whose first (marked) term is t' is the variant of `match_surj_last`. This marked triplet never appears again because the variant of the rule `match_surj_next` just increments the surjection rank of remaining triplets, whose first marked term t'' is another subterm of t, located at another position in t (t'' at least differs from t' by the name of its root symbol), and the other rules of $\mathscr{R}_1^M \cup \mathscr{R}_2^M$ neither create new marked triplets nor duplicate existing ones. □

Lemma 4.3 *The rewriting system $\mathscr{R}_1 \cup \mathscr{R}_2$ is terminating.*

Proof. Notice that $\mathscr{R}_1 \setminus \{\texttt{match\_AC}\}$ is clearly terminating since it is a subsytem of the eager one, which is known to be terminating. We deduce that $\mathscr{R}_1$ is terminating since no rule in $\mathscr{R}_1$ reduces the right side of the rule `match_AC`. Since $\mathscr{R}_1$ and $\mathscr{R}_2$ are terminating, it remains to show that there is no infinite reduction that goes back and forth between $\mathscr{R}_1$ and $\mathscr{R}_2$. Toward a contradiction, assume that there is an infinite reduction $C_1 \rightsquigarrow^+_{\mathscr{R}_1} C_2 \rightsquigarrow^+_{\mathscr{R}_2} C_3 \rightsquigarrow^+_{\mathscr{R}_1} \ldots$ that goes back and forth between $\mathscr{R}_1$ and $\mathscr{R}_2$, where C_1 is a pattern-matching problem. In $\mathscr{R}_2$ the rule `match_surj_next` is the only rule producing new redexes for the system $\mathscr{R}_1$, i.e. pattern-matching problems. Then the rule `match_surj_next` should appear infinitely often in this infinite reduction. Equivalently, we consider the marked variant of this derivation, with the same notations. Since the number of marked triplets in the sequence $C_1, C_2, \ldots$ is strongly bounded (by Lemma 4.2) there is a marked triplet (t,u,s_{k_i}) and an infinite sub-sequence $C_{k_1}, C_{k_2}, \ldots$ of $C_1, C_2, \ldots$ such that $\langle t,u,s_{k_i}\rangle \in C_{k_i}$ and $s_{k_{i+1}} = s_{k_i} + 1$ for each i. This is a contradiction since the rank of surjections is upper bounded. □

Theorem 4.4 *The Lazy system is terminating.*

Proof. On the one hand, the termination of $\mathscr{R}_1 \cup \mathscr{R}_2$ is proved in Lemma 4.3. On the other hand, it is standard to show that the system $\mathscr{R}_3$ is terminating. It corresponds to the normalization w.r.t. to the disjunctive normal form. Therefore, it is sufficient to show that there is no infinite reduction that goes back and forth between the saturation of $\mathscr{R}_1 \cup \mathscr{R}_2$ and the saturation of $\mathscr{R}_3$. Let $\mathcal{R} = t \ll_{AC} u \rightsquigarrow^{\omega}_{\mathscr{R}_1 \cup \mathscr{R}_2} C_1 \rightsquigarrow^{\omega}_{\mathscr{R}_3} C_2 \rightsquigarrow^{\omega}_{\mathscr{R}_1 \cup \mathscr{R}_2} \dots$ be a reduction in *Lazy*. Notice that each C_p, where p is even, is of the form $\bigvee_{i=1}^{i=k} F_i$ where $F_i = \bigwedge_{j=1}^{j=m} D_j$ and each D_j is either a triplet or the matching problem of a term with a variable. If there is no new redex in C_p then the reduction $\mathcal{R}$ stops. Otherwise, by observing the right side of the rules of $\mathscr{R}_1 \cup \mathscr{R}_2$, we claim that if there is a new redex in C_p – which is created by the system $\mathscr{R}_3$ – then this redex is necessarily of the form $x \ll_{AC} t \wedge \dots \wedge x \ll_{AC} t'$ with $t \neq t'$, producing the $\mathbf{F}$ constraint. Let $q \in [k]$ be the smallest integer such that such a redex appears in F_q. If $q = k$ then C_p is reduced to $\bigvee_{i=1}^{i=k-1} F_i \vee \mathbf{F}$ and the reduction $\mathcal{R}$ terminates. Otherwise, $C_p \rightsquigarrow_{\texttt{fail\_gen}} F_1 \vee \dots \vee \mathbf{F} \vee F_{q+1} \vee \dots \vee \mathbf{F} \vee \dots \vee F_k \rightsquigarrow_{\texttt{fail\_next}} F_1 \vee \dots \vee \mathit{Next}(F_{q+1}) \vee \dots \vee \mathbf{F} \vee \dots \vee F_k$.

The rules of $\mathscr{R}_2$ push the *Next* constructor down, and all the constraints of the form $\mathit{Next}(\langle t,u,s \rangle)$ in F_{p+1} are reduced to $(\bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i) \vee \langle t,u,s+1 \rangle$ by the rule `match_surj_next`, if $s(t) = t_\varepsilon(\alpha_1, \dots, \alpha_k)$. To prove that the reduction $\mathcal{R}$ terminates it is sufficient to prove that the system $\mathscr{R} = \mathscr{R}_1 \setminus \{\texttt{match\_AC}\} \cup \{\texttt{always\_next}, \texttt{match\_AC\_2}\} \cup \mathscr{R}_3$ is terminating, with the following rule definitions:

$$\texttt{match\_AC\_2:} \quad t \ll_{AC} u \rightsquigarrow \langle t,u,1 \rangle \quad \text{and} \quad \texttt{always\_next:} \quad \langle t,u,s \rangle \rightsquigarrow (\bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i) \vee \langle t,u,s+1 \rangle.$$

However the termination of $\mathscr{R}$ is ensured by the termination of the eager system. That is, we have just replaced the rule `E_match_AC` of *Eager* with the rules `match_AC_2` and `always_next` in $\mathscr{R}$. □

4.4 Confluence of lazy AC-matching

The system *Lazy* is not confluent, due to the non-confluence of $\mathscr{R}_3$. In this section we argue that the system $\mathscr{R}_1 \cup \mathscr{R}_2$ is confluent, and we consider an evaluation strategy for $\mathscr{R}_3$ to get a confluent AC-lazy matching system, that we call $\mathit{Lazy}^{\downarrow}$.

Proposition 4.5 *The system $\mathscr{R}_1 \cup \mathscr{R}_2$ is locally confluent.*

Proof. It is straightforward to check that there is no critical overlap between any two redexes, i.e. the contraction of one redex does not destroy the others. It is worth mentioning that without the condition $C_1 \neq \mathbf{F}$ of the rule `next_or` we could have non-convergent critical pairs, e.g. $\mathit{Next}(\mathbf{F} \vee C) \rightsquigarrow \mathit{Next}(\mathit{Next}(C))$ by the rule `fail_next` and $\mathit{Next}(\mathbf{F} \vee C) \rightsquigarrow \mathit{Next}(\mathbf{F}) \vee C \rightsquigarrow \mathbf{F} \vee C \rightsquigarrow \mathit{Next}(C)$. □

Corollary 4.6 *The system $\mathscr{R}_1 \cup \mathscr{R}_2$ is confluent.*

The reason of the non-confluence of $\mathscr{R}_3$ is the non-commutativity of the operators $\wedge$ and $\vee$. It is classical to add a strategy to $\mathscr{R}_3$ so that the resulting system becomes confluent.

Definition 4.2 Let $\mathscr{R}_3^{\downarrow}$ be the system $\mathscr{R}_3$ with the following strategy: *(i)* When reducing a constraint of the form $\bigwedge_{i=1}^{i=k} \bigvee_j C_{i,j}$ with $k \geq 3$, first reduce $\bigwedge_{i=2}^{i=k} \bigvee_j C_{i,j}$, and *(ii)* reduce $(\bigvee_{l=1}^{l=m} A_l) \wedge B$ to $(A_1 \wedge B) \vee ((\bigvee_{l=2}^{l=m} A_l) \wedge B)$.

Proposition 4.7 *(Admitted)* $\mathscr{R}_3^{\downarrow}$ *is confluent.*

Now we are ready to define the lazy AC-matching.

Definition 4.3 The *lazy AC-matching*, denoted by $Lazy^{\downarrow}$, is the rewriting system $\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3$ composed of the rules of Figures 2, 3 and 4, equipped with the evaluation strategy $sat^{\downarrow}$ that consists of the iteration of the following process: *(i)* Applying the rules of $\mathscr{R}_1 \cup \mathscr{R}_2$ until no rule is applicable, then *(ii)* applying $\mathscr{R}_3^{\downarrow}$ until no rule is applicable.

Theorem 4.8 $Lazy^{\downarrow}$ *is terminating and confluent.*

Proof. The termination of $Lazy^{\downarrow}$ is a consequence of the one of *Lazy*. The confluence of $Lazy^{\downarrow}$ follows from the confluence of $\mathscr{R}_1 \cup \mathscr{R}_2$ and $\mathscr{R}_3^{\downarrow}$. □

In what follows the normal form of a constraint C w.r.t. a system $\mathscr{R}$ will be denoted by $NF_{\mathscr{R}}(C)$, or just $NF(C)$ if $\mathscr{R}$ is $Lazy^{\downarrow}$.

4.5 Normal forms and lazy lists

In this section we prove Theorem 4.15 that characterizes the normal forms of the lazy AC-matching $Lazy^{\downarrow}$. They correspond basically to lazy lists. Roughly speaking, a lazy list is composed of a substitution at the head and a non-evaluated object that represents the remaining substitutions. This characterization of the normal forms is of major importance since it guarantees that the element at the head is always a substitution. The formal definition of lazy lists follows.

Definition 4.4 A $\wedge$-*substitution* is a conjunction of delayed matching constraints of the form $x \ll_{AC} u$ where x is a variable. A $\wedge$-substitution is *irreducible* if it cannot be reduced by the rule `fail_gen`. A constraint is called a *lazy list* if it is $\mathbf{F}$, $\mathbf{I}$, an irreducible $\wedge$-substitution or a constraint of the form $\sigma \vee C$ where σ is an irreducible $\wedge$-substitution and $NF(Next(C))$ is also a lazy list.

In order to characterize the normal forms of $Lazy^{\downarrow}$ we first characterize in Lemma 4.9 the normal forms of the system $\mathscr{R}_1 \cup \mathscr{R}_2$. Then we characterize in Lemma 4.12 the normal forms of $\mathscr{R}_3^{\downarrow}$ when it has the normal forms of $\mathscr{R}_1 \cup \mathscr{R}_2$ as input. Summing up these results, we show in Proposition 4.14 the invariance of the syntax of constraints after the composition of the application of $\mathscr{R}_1 \cup \mathscr{R}_2$ and of $\mathscr{R}_3^{\downarrow}$. Finally, Theorem 4.15 becomes an immediate consequence of Proposition 4.14.

Let us begin by characterizing the normal forms of $\mathscr{R}_1 \cup \mathscr{R}_2$ and introduce for this purpose the grammar $\mathcal{G} ::= \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \langle \mathcal{T}, \mathcal{T}, \mathbb{N}^* \rangle \mid \mathscr{X} \ll_{AC} \mathcal{T} \mid \mathbf{I}$.

Lemma 4.9 *The normal form of an AC-matching problem by the system* $\mathscr{R}_1 \cup \mathscr{R}_2$ *is either* $\mathbf{F}$ *or follows the grammar* $\mathcal{G}$.

Proof. Let $t \ll_{AC} u$ be an AC-matching problem. When $t \ll_{AC} u \rightsquigarrow_{\mathscr{R}_1 \cup \mathscr{R}_2} \mathbf{F}$, the normal form of the AC-matching problem is $\mathbf{F}$, since no rule rewrites $\mathbf{F}$. The other cases when t or u is a variable are also trivial: The matching problem is reduced to $\mathbf{I}$ or is irreducible.

It remains to consider the case when $t = t_\varepsilon(t_1, \ldots, t_k)$ and $u = t_\varepsilon(u_1, \ldots, u_n)$ for some $k, n \geq 0$. The proof is by induction on the number of symbols in t. If t is a constant, i.e. $k = 0$, then $n = 0$ and $u = t$. The AC-matching problem $t \ll_{AC} u$ is reduced to $\mathbf{I}$, thus follows the grammar $\mathcal{G}$. Otherwise, $k \geq 1$. Only one rule can be applied, `match_AC` or `match`, depending on the nature of the symbol t_ε at the root of t.

Case 1. If t_ε is an AC symbol, then $k \leq n$ and $t \ll_{AC} u \rightsquigarrow_{\texttt{match\_AC}} Next(\langle t,u,1\rangle)$. We prove more generally that the normal form of $Next(\langle t,u,s\rangle)$ by $\mathscr{R}_1 \cup \mathscr{R}_2$ is $\mathbf{F}$ or follows the grammar $\mathcal{G}$. Let $s(u) = t_\varepsilon(\alpha_1,\ldots,\alpha_k)$ and $C = \bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i$.

If $|S_{n,k}| - s = 0$, then $Next(\langle t,u,s\rangle) \rightsquigarrow_{\texttt{match\_surj\_last}} C \rightsquigarrow^\star NF(C)$. Otherwise, when $|S_{n,k}| - s > 0$, $Next(\langle t,u,s\rangle) \rightsquigarrow_{\texttt{match\_surj\_next}} C \vee \langle t,u,s+1\rangle \rightsquigarrow^\star NF(C) \vee \langle t,u,s+1\rangle$. In both cases, since each t_i contains less symbols than t, the induction hypothesis holds for each t_i, and hence the normal form $NF(C)$ of $C = \bigwedge_{i=1}^{i=k} t_i \ll_{AC} \alpha_i$ is $\mathbf{F}$ or a constraint which follows the grammar $\mathcal{G}$, since $\mathbf{F}$ is an absorbing element for $\wedge$.

For this case the remainder of the proof is by induction on $|S_{n,k}| - s$. The basic case when $|S_{n,k}| - s = 0$ has already been treated. When $|S_{n,k}| - s > 0$, there are two cases. If $NF(C)$ follows the grammar $\mathcal{G}$ then it obviously also holds for $NF(C) \vee \langle t,u,s+1\rangle$. Otherwise, $NF(C)$ is $\mathbf{F}$ and $Next(\langle t,u,s\rangle) \rightsquigarrow^\star \mathbf{F} \vee \langle t,u,s+1\rangle \rightsquigarrow_{\texttt{fail\_next}} Next(\langle t,u,s+1\rangle)$. Since $|S_{n,k}| - (s+1) < |S_{n,k}| - s$, the induction hypothesis gives that $NF(Next(\langle t,u,s+1\rangle))$ is $\mathbf{F}$ or follows the grammar $\mathcal{G}$, hence also for $Next(\langle t,u,s\rangle)$.
Case 2. If t_ε is not an AC symbol, then $k = n$ and $t \ll_{AC} u \rightsquigarrow_{\texttt{match}} \bigwedge_{i=1}^{i=k} t_i \ll_{AC} u_i \rightsquigarrow^\star \bigwedge_{i=1}^{i=k} NF(t_i \ll_{AC} u_i)$. By induction hypothesis, for each i, $NF(t_i \ll_{AC} u_i)$ is $\mathbf{F}$ or follows the grammar $\mathcal{G}$. If $NF(t_i \ll_{AC} u_i) = \mathbf{F}$ for some $i \in [k]$, then $t \ll_{AC} u \rightsquigarrow^* \mathbf{F}$, since $\mathbf{F}$ is an absorbing element for $\wedge$. Otherwise, $NF(t_i \ll_{AC} u_i)$ follows $\mathcal{G}$ for each i, and then it obviously also holds for their conjunction, and for $NF(t \ll_{AC} u)$. □

Lemma 4.10 *Let C be an irreducible constraint w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$ that follows the grammar $\mathcal{G}$. Then, the normal form of $Next(C)$ by $\mathscr{R}_1 \cup \mathscr{R}_2$ is C.*

Proof. The proof is by induction on the grammar constructions of $\mathcal{G}$. If C is $\mathbf{I}$ or a matching problem of the form $x \ll_{AC} u$ where x is a variable, then the rules `next_id` and `next_basic` ensure that $Next(C) \rightsquigarrow C$. If $C = C_1 \wedge C_2$ then $Next(C_1 \wedge C_2) \rightsquigarrow Next(C_1) \wedge Next(C_2)$ and the induction hypothesis $NF(Next(C_1)) = C_1$ and $NF(Next(C_2)) = C_2$ apply to show that $NF(Next(C_1 \wedge C_2)) = NF(C_1 \wedge C_2) = C_1 \wedge C_2$. If $C = C_1 \vee \langle t,u,s\rangle$, then $Next(C_1 \vee \langle t,u,s\rangle) \rightsquigarrow Next(C_1) \vee \langle t,u,s\rangle \rightsquigarrow C_1 \vee \langle t,u,s\rangle$. □

We can generalize the previous lemma by the following one.

Lemma 4.11 *Let F be a constraint of the form $\bigwedge_i F_i$, where each F_i is either a triplet or follows the grammar $\mathcal{G}$, such that F is irreducible w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$. Then, the normal form of $Next(F)$ by $\mathscr{R}_1 \cup \mathscr{R}_2$ is $\mathbf{F}$ or follows the grammar $\mathcal{G}$.*

Proof. By iterating application of the rule `next_and` we get $Next(\bigwedge_i F_i) \rightsquigarrow^* \bigwedge_i Next(F_i)$. F_i is irreducible w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$ because F is irreducible. If F_i follows the grammar $\mathcal{G}$, then $NF(Next(F_i)) = F_i$ by Lemma 4.10. If F_i is a triplet then by Lemma 4.9 the normal form of $Next(F_i)$ is $\mathbf{F}$ or follows $\mathcal{G}$. Therefore, the normal form of $\bigwedge_i Next(F_i)$ is $\mathbf{F}$ or follows the grammar $\mathcal{G}$. □

We define the grammar $\mathcal{K} ::= \mathcal{K} \wedge \mathcal{K} \mid \mathcal{X} \ll_{AC} \mathcal{T} \mid \langle \mathcal{T}, \mathcal{T}, \mathbb{N}^\star \rangle$ for conjunctions of atomic constraints, the grammar $\mathcal{F} ::= \mathcal{F} \vee \mathcal{F} \mid \mathcal{K}$ for constraints in DNF, the grammar $\mathcal{S} ::= \mathcal{S} \wedge \mathcal{S} \mid \mathcal{X} \ll_{AC} \mathcal{T}$ for $\wedge$-substitutions, and the grammar $\mathcal{H} ::= \mathcal{S} \vee \mathcal{F} \mid \mathcal{S} \mid \mathbf{I} \vee \mathcal{F} \mid \mathbf{I}$ to formulate the following two lemmas. The first one is about normal forms by $\mathscr{R}_3^\downarrow$ of inputs which are normal forms of $\mathscr{R}_1 \cup \mathscr{R}_2$.

Lemma 4.12 *Let C be an irreducible constraint w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$ that follows the grammar $\mathcal{G}$. Then, the normal form of C by the system $\mathscr{R}_3^\downarrow$ follows the grammar $\mathcal{H}$.*

Proof. On the one hand, since C follows the grammar $\mathcal{G}$, it is built up on $\wedge$, $\vee$, $\mathbf{I}$, matching constraints of the form $x \ll_{AC} u$ and triplets. On the other hand, the normal form of C by $\mathscr{R}_3^{\downarrow}$ is in DNF. Therefore it is sufficient to show that $NF_{\mathscr{R}_3^{\downarrow}}(C)$ is either $\mathbf{I}$ or of the form $\sigma \vee F$, where σ is either a $\wedge$-substitution or $\mathbf{I}$, and F follows $\mathcal{F}$. The proof is by induction on the grammar constructions of $\mathcal{G}$. If C is $\mathbf{I}$ or $x \ll_{AC} u$ then the claim holds. Otherwise, we distinguish two cases:
Case 1. If $C = C_1 \wedge C_2$, then we only discuss the non-trivial case when $NF(C_1)$ or $NF(C_2)$ is of the form $\sigma \vee F$. Assume that $C_1 = \sigma_1 \vee F_1$, the other case can be handled similarly. In this case C_2 can be $\mathbf{I}$, σ_2 or $\sigma_2 \vee F_2$. If $C_2 = \mathbf{I}$ then $(\sigma_1 \vee F_1) \wedge \mathbf{I} = \sigma \vee F_1$. If $C_2 = \sigma_2$, then $(\sigma_1 \vee F_1) \wedge \sigma_2 \rightsquigarrow (\sigma_1 \wedge \sigma_2) \vee (F_1 \wedge \sigma_2)$. Finally, if $C_2 = \sigma_2 \vee F_2$, then $(\sigma_1 \vee F_1) \wedge (\sigma_2 \vee F_2) \rightsquigarrow^* (\sigma_1 \wedge \sigma_2) \vee (\sigma_1 \wedge F_2) \vee (F_1 \wedge \sigma_2) \vee (F_1 \wedge F_2)$ and the claim holds.
Case 2. If $C = C_1 \vee \langle t, u, s \rangle$, then we apply the induction hypothesis on $NF(C_1)$, and the desired result follows. □

The following lemma describes the syntax of the result of $Next(F)$ by the application of $\mathscr{R}_1 \cup \mathscr{R}_2$ followed by the application of $\mathscr{R}_3^{\downarrow}$, when F is an irreducible constraint in DNF.

Lemma 4.13 *Let $\phi(p) = \bigvee_{i=1}^{i=p} \bigwedge_{j=1}^{j=q} C_{i,j}$ be an irreducible constraint w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$ that follows the grammar $\mathcal{F}$. Then $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(\phi(p))))$ is either $\mathbf{F}$ or follows the grammar $\mathcal{H}$.*

Proof. The proof is by induction on p. If $p = 1$ then by Lemma 4.11 $\phi' = NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(\phi(1)))$ is $\mathbf{F}$ or follows $\mathcal{G}$. Therefore $NF_{\mathscr{R}_3^{\downarrow}}(\phi')$ is $\mathbf{F}$, or follows $\mathcal{H}$ by Lemma 4.12. If $p > 1$ then $Next(\bigvee_{i=1}^{i=p} \bigwedge_{j=1}^{j=q} C_{i,j}) \rightsquigarrow Next(\bigwedge_{j=1}^{j=q} C_{1,j}) \vee \phi(p-1)$. By Lemma 4.11 again $NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(\bigwedge_{j=1}^{j=q} C_{1,j}))$ is $\mathbf{F}$ or follows $\mathcal{G}$.

In the first case, we apply $\mathbf{F} \vee \phi(p-1) \rightsquigarrow Next(\phi(p-1))$, and use the induction hypothesis that $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(\phi(p-1))))$ follows $\mathcal{H}$. In the second case, it comes from Lemma 4.12 that $\psi = NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(\bigwedge_{j=1}^{j=q} C_{1,j})))$ follows $\mathcal{H}$. Hence $\psi \vee (\phi(p-1))$ follows $\mathcal{H}$, since $\phi(p-1)$ follows $\mathcal{F}$. □

Now we are ready to prove the following invariance proposition. It generalizes the previous lemma by considering an arbitrary constraint following the grammar $\mathcal{H}$.

Proposition 4.14 *(Invariance proposition) Let C be a constraint following the grammar $\mathcal{H}$. Then $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(C))$ is $\mathbf{F}$ or follows the grammar $\mathcal{H}$.*

Proof. The case when C is a $\wedge$-substitution or $\mathbf{I}$ is trivial. Otherwise, let $C = \sigma \vee F$, where σ is $\mathbf{I}$ or a $\wedge$-substitution and F follows $\mathcal{F}$. Notice that the only potential redexes in C w.r.t. the system $\mathscr{R}_1 \cup \mathscr{R}_2$ are of the form $x \ll_{AC} u_1 \wedge \ldots \wedge x \ll_{AC} u_2$ such that $u_1 \neq u_2$. In this case the rule `fail_gen` is applied. Let us call such redexes *failure redexes.* We distinguish two cases.
Case 1. If there is no failure redex in F (i.e. F is irreducible w.r.t. $\mathscr{R}_1 \cup \mathscr{R}_2$) then we again distinguish two cases. If there is no failure redex in σ, then we are done. Otherwise, $\sigma \vee F \rightsquigarrow \mathbf{F} \vee F \rightsquigarrow Next(F)$, and the result follows from Lemma 4.13.
Case 2. If there are some failure redexes in F then assume that $F = \bigvee_{i=1}^{i=m} F_i$, and let $I = [m]$. Let us argue that $NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(F)$ is either $\mathbf{F}$ or of the form $\bigvee_{i \in I'} F_i'$ where either $F_i' = F_i$ or $F_i' = NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(Next(F_i)) \neq \mathbf{F}$ for some $I' \subseteq I$. We propose an algorithm to construct I'. Let $I' := I$ initially. (a) If $m = 1$, the expected form for $NF_{\mathscr{R}_1 \cup \mathscr{R}_2}(F)$ is obtained with the current I'. Otherwise, if $m > 1$, let p be the smallest integer in $[m]$ such that F_i contains a failure redex. Therefore we have $\bigvee_{i=1}^{i=m} F_i \rightsquigarrow \bigvee_{i=1}^{i=p-1} F_i \vee$ `fail` $\vee$

$F_{p+1} \vee \bigvee_{i=p+2}^{i=m} F_i \rightsquigarrow \bigvee_{i=1}^{i=p-1} F_i \vee Next(F_{p+1}) \vee \bigvee_{i=p+2}^{i=m} F_i$. Continue the elimination of the failure redexes in $\bigvee_{i=p+2}^{i=m} F_i$ by iterating (a) with $\{p+2,\dots,m\}$ instead of $[m]$. Let G by the resulting disjunction. If $NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(F_{p+1})) \neq \mathbf{F}$, then let $I' := I' \setminus \{p\}$. Otherwise, if $NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(F_{p+1})) = \mathbf{F}$, then let $I' := I' \setminus \{p, p+1\}$ and continue the reduction on $Next(G)$.

Since $F'_i = F_i$ or $F'_i = NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(F_i))$, where $i \in I'$, then

(i) $NF_{\mathscr{R}_3^{\downarrow}}(F'_i)$ either follows $\mathcal{F}$ or follows $\mathcal{H}$ by Lemma 4.12. Therefore, $NF_{\mathscr{R}_3^{\downarrow}}(\bigvee_{i\in I'} F'_i)$ is either $\mathbf{F}$ or follows $\mathcal{F}$ and $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(F))$ is either $\mathbf{F}$ or follows the grammar $\mathcal{F}$.

(ii) In order to simplify the computations, let $q = |I'|$ and consider the renaming $\bigvee_{i\in[q]} H_i = \bigvee_{i\in I'} F'_i$. We argue by induction on q that $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(\bigvee_{i\in[q]} H_i)))$ is either $\mathbf{F}$ or follows $\mathcal{H}$. If $q = 1$, then by Lemma 4.11 it follows that $NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(H_i))$ is either $\mathbf{F}$ or follows $\mathcal{G}$ and therefore, by Lemma 4.12, we have that $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(H_i)))$ is either $\mathbf{F}$ or follows $\mathcal{H}$. If $q > 1$, then $Next(\bigvee_{i\in[q]} H_i) \rightsquigarrow Next(H_1) \vee \bigvee_{i=2}^{i=q} H_i$. If $Next(H_1) \rightsquigarrow^\star \mathbf{F}$, then we apply the induction hypothesis to $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(\bigvee_{i=2}^{i=q} H_i)))$. Otherwise, $NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(H_1))$ follows $\mathcal{G}$, and hence $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(H_1)))$ follows $\mathcal{H}$. On the other hand, $NF_{\mathscr{R}_3^{\downarrow}}(\bigvee_{i=2}^{q} H_i)$ follows $\mathcal{F}$, by (i). Summing up, we get that $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(H_1))) \vee NF_{\mathscr{R}_3^{\downarrow}}(\bigvee_{i=2}^{q} H_i))$ follows $\mathcal{H}$.

Now we distinguish two cases for σ. If there is no failure redex in σ, then by (i) we get that $\sigma \vee NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(F))$ is either a $\wedge$-substitution or follows $\mathcal{H}$. Otherwise, if there are some failure redexes in σ, then we get $\sigma \vee F \rightsquigarrow^\star \mathbf{F} \vee NF_{\mathscr{R}_1\cup\mathscr{R}_2}(F) \rightsquigarrow Next(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(F))$. From (ii) it follows that $NF_{\mathscr{R}_3^{\downarrow}}(Next(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(F)))$ is $\mathbf{F}$ or follows $\mathcal{H}$. □

Theorem 4.15 *The normal form of a pattern-matching constraint C by the system $Lazy^{\downarrow}$ is a lazy list.*

Proof. From the termination of $Lazy^{\downarrow}$ (Theorem 4.8) and Proposition 4.14, we deduce that the normal form of $Lazy^{\downarrow}$ is either $\mathbf{F}$ or follows the grammar $\mathcal{H}$ and does not contain any failure redex. Such a normal form is of the form σ or $\sigma \vee F$, where σ is either $\mathbf{I}$ or an irreducible $\wedge$-substitution. Therefore, it remains to show that $NF(Next(F))$ is a lazy list, or, equivalently, that $NF_{\mathscr{R}_3^{\downarrow}}(NF_{\mathscr{R}_1\cup\mathscr{R}_2}(Next(F)))$ follows $\mathcal{H}$. But this holds by Lemma 4.13. □

5 Lazy AC-rewriting with strategies

In this section we integrate lazy AC-matching with strategy application. More details on strategy languages can be found in [3, 15, 13].

Primitive strategies are rewrite rules $l \to r$ and the id and fail strategies that respectively always and never succeed. They are completed with the most usual reduction strategies, namely the four traversal strategies *leftmost-outermost*, *leftmost-innermost*, *parallel-outermost* and *parallel-innermost* [14, Definition 4.9.5] that control a rewrite system by selecting redexes according to their position. For sake of simplicity we restrict their control to a single rewrite rule. Let v be one of these four strategies. The application of v to the rewrite rule $l \to r$ is denoted by $v(l \to r)$. The sequential composition of two strategies u and w is denoted by $u;w$. The application of a strategy u to a term t is denoted by $[u]\cdot t$.

A strategy application produces a lazy list of terms, defined by the grammar $\mathcal{L} ::= \bot_{\mathcal{T}} \mid \mathcal{L} :: \mathcal{L} \mid \mathcal{T} \mid \mathcal{C}(\mathcal{T})$. A list is usually defined by a constructor for an empty list and a constructor adding one element at the head of another list. Then concatenation of two lists is defined, with another notation. Here we

equivalently introduce an associative symbol :: for concatenation of two lists of terms, and a symbol $\bot_{\mathcal{T}}$ to denote an empty list of terms, which is a neutral element for ::. We use the same conventions as for $\vee$ in Section 4. Let $LList(\mathcal{T})$ denote the set of lazy lists of terms.

identity:	$[\mathsf{id}]\cdot\tau \rightsquigarrow \tau$
failure:	$[\mathsf{fail}]\cdot\tau \rightsquigarrow \bot_{\mathcal{T}}$
compose:	$[u;v]\cdot\tau \rightsquigarrow [u]\cdot([v]\cdot\tau)$

(a) id, fail and composition rules

rule1:	$[l\to r]\cdot\bot_{\mathcal{T}} \rightsquigarrow \bot_{\mathcal{T}}$
rule2:	$[l\to r]\cdot(t :: \tau) \rightsquigarrow (l\ll_{AC} t)(r) :: ([l\to r]\cdot\tau)$
subs_fail:	$\mathbf{F}(t) \rightsquigarrow \bot_{\mathcal{T}}$
subs_id:	$\mathbf{I}(t) \rightsquigarrow t$
subs:	$(\sigma\vee C)(t) \rightsquigarrow \sigma(t) :: C(t)$

(b) Top rewriting

Figure 5: AC-rewriting operational semantics

The operational semantics of the strategies id and fail and of strategy composition are defined in Figure 5(a) for any lazy list of terms τ and any two strategies u and v. The operational semantics of top rewriting is defined in Figure 5(b). Let LTR be the system composed of these five rules and the $Lazy^{\downarrow}$ AC-matching. The rules rule1 and rule2 reduce the application of a rewrite rule at the top of terms in a lazy list of terms. In rule2 the expression $l\ll_{AC} t$ is reduced to its normal form by $Lazy^{\downarrow}$. The result is a lazy list of constraints. The rules subs_fail, subs_id and subs reduce the application of a lazy list of constraints on a term. In the right side of rule subs, a $\wedge$-substitution σ is applied to a term, in a sense which is a simple extension of the standard definition of substitution application. Equivalently $\wedge$-substitutions can be reduced to standard ones by a transformation similar to the post-processing defined for the eager AC-matching.

The application of the system LTR to the expression $[l\to r]\cdot(t)$ produces either $\bot_{\mathcal{T}}$ or a non-empty lazy list of terms $u_1 :: \tau_1$, where u_1 is the first result of the application of the rewrite rule $l\to r$ at the top of the term t and τ_1 is (a syntactic object denoting) the lazy list of the other results. When applying on τ_1 the rewrite system defined in Figure 6, and then the system LTR, we get again either $\bot_{\mathcal{T}}$ or a non-empty lazy list $u_2 :: \tau_2$, where u_2 is the second result of the application of the rewrite rule $l\to r$ at the top of the term t and τ_2 is again (a syntactic object denoting) a lazy list of terms that represents the remaining results, and so on.

next_empty:	$Next(\bot_{\mathcal{T}}) \rightsquigarrow \bot_{\mathcal{T}}$	next_app:	$Next(L_1 :: L_2) \rightsquigarrow Next(L_1) :: L_2$
next_term:	$Next(t) \rightsquigarrow t$, if t is a term	next_cstr:	$Next(C(t)) \rightsquigarrow Next(C)(t)$

Figure 6: *Next* rules for lists of terms

The application of a traversal strategy on a lazy list of terms in $LList(\mathcal{T})$ is defined as follows:

traversal1:	$[v(l\to r)]\cdot\bot_{\mathcal{T}} \rightsquigarrow \bot_{\mathcal{T}}$
traversal2:	$[v(l\to r)]\cdot(t :: \tau) \rightsquigarrow [v(l\to r)]\cdot t :: [v(l\to r)]\cdot\tau$

The rules

$$[v(u)]\cdot t \rightsquigarrow \begin{cases} [u]\cdot t & \text{if } [u]\cdot t \neq \bot_{\mathcal{T}} \text{ or } t\in\mathcal{X} \\ \uparrow f([v(u)]\cdot t_1,\ldots,[v(u)]\cdot t_n) & \text{if } [u]\cdot t = \bot_{\mathcal{T}} \text{ and } t = f(t_1,\ldots,t_n) \end{cases}$$

define the application of the traversal strategy $v(u)$ on the term t for the rewrite rule u and the *parallel-outermost* strategy constructor v. The other traversal strategies can be handled similarly. We have seen that the application of a rewrite rule at the top of a term yields a lazy list of terms in $LList(\mathcal{T})$. Here

the application of a rule to a term at arbitrary depth, via a traversal strategy, yields a *decorated* term, which is a term where some subterms are replaced by a lazy list of terms. This lazy list of terms is abusively called a *lazy subterm*, with the property that lazy subterms are not nested. In other words, the positions of two lazy subterms are not comparable, for the standard prefix partial order over the set of term positions. The operator $\uparrow$ is assumed to reduce a decorated term to a lazy list of terms. We summarize its behavior as follows. A decorated term can be encoded by a tuple (t,k,p,δ) where t is the term before strategy application, k is the number of decorated positions, p is a function from $\{1,\ldots,k\}$ to the domain of t (i.e. its set of positions) which defines the decorated positions, such that $p(i)$ and $p(j)$ are not comparable if $i \neq j$, and δ is the function from $\{1,\ldots,k\}$ to $LList(\mathcal{T})$ such that $\delta(i)$ is the lazy list at position $p(i)$, $1 \leq i \leq k$. It is easy to construct an iterator over the k-tuples of positive integers (for instance in lexicographical order), and to derive from it an iterator over tuples of terms $(s_1,\ldots,s_k)$ with s_i in the list $\delta(i)$ for $1 \leq i \leq k$. From this iterator and function p we derive an iterator producing the lazy list $\uparrow t$ by replacing each subterm $t_{|p(i)}$ by s_i.

6 Implementation and experiments

We present here a prototypal implementation of lazy AC-matching and report about its experimentation. Our implementation is a straightforward translation of the $Lazy^{\downarrow}$ system in the rule-based language `symbtrans` [1] built on the computer algebra system Maple.

This prototype obviously does not claim efficiency in the usual sense of the number of solutions computed in a given amount of time. But this section shows that our prototype optimises the standard deviation of the time between two successive solutions. This performance criterion corresponds to our initial motivations and can be measured on the prototype. We consider the matching problem $x_1 + \ldots + x_{18} \ll_{AC} a_1 + \ldots + a_{18}$, for 18 variables x_1, ..., x_{18} and 18 constants a_1, ..., a_{18}. On this problem our lazy prototype provides any two consecutive solutions in an average time of 0.37 seconds, with a standard deviation of 0.021 seconds between the 100-th first solutions. In comparison the computation time between two consecutive solutions with the Maude function `metaXmatch` grows exponentially. The experiment is done on an Intel core 2 Duo T6600@2.2GHz with 3.4Gb of memory, under a x86_64 Ubuntu Linux.

Finally, it is worth mentioning the performance of the Maple standard matching procedure `pat-match(expr, pattern,'s')` that returns true if it is able to match `expr` to `pattern`, and false otherwise. If the matching is successful, then `s` is assigned a substitution such that $AC \models$ `s(pattern)=expr`. This procedure runs out of memory if the arity of the AC symbols is large. With Maple 14 this failure can be observed when computing a solution of the matching problem $x_1 + \ldots + x_{12} \ll_{AC} a_1 + \ldots + a_{12}$.

7 Conclusion

We presented a lazy AC-matching algorithm and a lazy evaluation semantics for AC-rewriting and some basic strategies. The semantics is designed to be implemented in a strict language by representing delayed matching constraints and lazy lists of terms by explicit objects. We also described a common principle for lazy traversal strategies. The potential benefits are clear: performances are dramatically increased by avoiding unnecessary computations. We are working on an implementation of lazy AC-matching and AC-rewriting: first results show that our approach is efficient when the arity of AC symbols is high and when the number of solutions of the AC-matching problem is large. However, we do not claim efficiency for the search of the first solution by the AC-matching algorithm.

Here no neutral element is assumed for AC symbols. As a consequence the lazy AC-matching relies on a surjection iterator. We plan to address the question of its efficiency, and to extend the present work to AC symbols with a neutral element. Our intuition is that our approach can easily be adapted to that case.

References

[1] W. Belkhir, A. Giorgetti & M. Lenczner (December 2010): *Rewriting and Symbolic Transformations for Multi-scale Methods*. Url: http://arxiv.org/abs/1101.3218v1. Submitted.

[2] D. Benanav, D. Kapur & P. Narendran (1985): *Complexity of matching problems*. In: *Proc. of the 1st Int. Conf. on Rewriting Techniques and Applications*, *LNCS* 202, Springer, pp. 417–429. doi:10.1007/3-540-15976-2_22.

[3] P. Borovanský, C. Kirchner, H. Kirchner & C. Ringeissen (2001): *Rewriting with strategies in* ELAN*: a functional semantics*. *International Journal of Foundations of Computer Science* 12(1), pp. 69–98. doi:10.1142/S0129054101000412.

[4] H. Cirstea, G. Faure & C. Kirchner (2007): *A ρ-calculus of explicit constraint application*. *Higher-Order and Symbolic Computation* 20, pp. 37–72. doi:10.1007/s10990-007-9004-2.

[5] H. Cirstea & C. Kirchner (2001): *The rewriting calculus — Part I* and *II*. *Logic Journal of the Interest Group in Pure and Applied Logics* 9(3), pp. 427–498.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *LNCS* 4350, Springer.

[7] N. Dershowitz (1982): *Ordering for Term-Rewriting Systems*. *Theoretical Computer Science* 17, pp. 279–300. doi:10.1016/0304-3975(82)90026-3.

[8] S. Eker (1995): *AC-Matching Via Bipartite Graph Matching*. *Comput. J.* 38(5), pp. 381–399. doi:10.1093/comjnl/38.5.381.

[9] S. Eker (1996): *Fast matching in combinations of regular equational theories*. *ENTCS* 4, pp. 90–109.

[10] S. Eker (2002): *Single Elementary AC-Matching*. *J. Autom. Reasoning* 28(1), pp. 35–51. doi:10.1023/A:1020122610698.

[11] B. Gramlich (1988): *Efficient AC-Matching using Constraint Propagation*. In: *Proc. 2nd Int. Workshop on Unification, Internal Report 89 R 38, CRIN*, Val d'Ajol, France.

[12] H. Kirchner & P.-E. Moreau (2001): *Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in AC-theories*. *J. Funct. Program.* 11, pp. 207–251.

[13] N. Martí-Oliet, J. Meseguer & A. Verdejo (2005): *Towards a Strategy Language for Maude*. *Electr. Notes Theor. Comput. Sci.* 117, pp. 417–441. doi:10.1016/j.entcs.2004.06.020.

[14] Terese (2003): *Term Rewriting Systems*. *Cambridge Tracts in Theor. Comp. Sci.* 55, Cambridge Univ. Press.

[15] E. Visser (2001): *Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5*. In: *Proc. of RTA'01*, *Lecture Notes in Computer Science* 2051, Springer-Verlag, pp. 357–361. doi:10.1007/3-540-45127-7_27.

[16] B. Yang, W. Belkhir, R.N. Dhara, M. Lenczner & A. Giorgetti (2011): *Computer–Aided Multiscale Model Derivation for MEMS Arrays*. In: *EuroSimE 2011, 13-th Int. Conf. on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems*, IEEE Computer Society, Linz, Austria. 6 pages. doi:10.1109/ESIME.2011.5765784.

[17] B. Yang, R.N. Dhara, W. Belkhir, M. Lenczner & A. Giorgetti (2011): *Formal Methods for Multiscale Models Derivation*. In: *CFM 2011, 20th Congrès Français de Mécanique*. 5 pages.