Variadic Equational Matching in Associative and Commutative Theories

Besik Dundua[a,b], Temur Kutsia[c], Mircea Marin[d]

[a] *Kutaisi International University, Georgia*
[b] *VIAM, Ivane Javakhishvili Tbilisi State University, Georgia*
[c] *RISC, Johannes Kepler University Linz, Austria*
[d] *West University of Timişoara, Romania*

Abstract

In this paper we study matching in equational theories that specify counterparts of associativity and commutativity for variadic function symbols. We design a procedure to solve a system of matching equations and prove its termination, soundness, completeness, and minimality. The minimal complete set of matchers for such a system can be infinite, but our algorithm computes its finite representation in the form of solved set. From the practical side, we identify two finitary cases and impose restrictions on the procedure to get an incomplete algorithm, which, based on our experiments, describes the input-output behavior and properties of Mathematica's flat and orderless pattern matching.[1]

Keywords: Pattern matching, equational theories, variadic symbols, Mathematica
2000 MSC: 03B70, 68Q42, 68N15, 33F10

1. Introduction

In variadic languages, function symbols do not have a fixed arity. They can take an arbitrary number of arguments. In the literature, such symbols are known by different names: flexary, of flexible arity, polyadic, multi-ary, unranked. They are a convenient and useful tool for formalizing mathematical texts, representing symbolic computation data structures, modeling

[1] The original experiments have been carried out for Mathematica 11.2. Later, they have been repeated for Mathematica 12.0 with the same results.

XML documents, expressing patterns in declarative programming, etc. Usually, variadic languages contain variables not only for individual terms, but also for finite sequences of terms, which help to take the full advantage of the flexibility of such languages.

On the other hand, the increased expressiveness of variadic languages has its price, from the computational perspective. Solving equations with sequence variables is a nontrivial task (Kutsia, 2002b, 2004, 2007) and pattern matching, a very common operation in the above-mentioned applications, becomes pretty involved.

In this paper we address the problem of pattern matching in variadic languages, where some function symbols satisfy (the variadic counterparts of) the commutativity (C) and associativity (A) properties. Equational matching in these theories has been intensively studied in languages with ranked alphabets, see, e.g., (Benanav et al., 1987; Eker, 2002, 2003; Hullot, 1979). Variadic equational matching so far attracted less attention.

We try to address this shortcoming, approaching the problem both from the theoretical and application points of view. From the theoretical side, we propose a modular rule-based system for solving matching equations with A, C, and AC function symbols. The major problem is that variadic A- and AC-matching problems might have infinitely many incomparable solutions. (They are examples of so called *infinitary* matching theories.) Hence, any procedure that aims at enumerating the minimal complete set of matchers in these theories is nonterminating. Therefore, we propose a finite representation of this infinite set. From this representation, there is an "easy way" to obtain any solution of the problem. It applies to both A- and AC-matching. Our rules give algorithms for such a finite representation in the corresponding theories. We prove termination, soundness and completeness of the algorithms.

Our focus was not on coming up with an optimized, efficient procedure. Rather, we chose a declarative, modular approach, which makes proving properties easier. From the application perspective, we show how some intuitive modifications of the rules can lead to finitary cases. Two such special cases: bounded fragment and strict variant, are studied in the paper.

The final part of the paper is devoted to the analysis of the behavior of the equational variadic matching algorithm implemented in the symbolic computation system Mathematica (Wolfram, 2003). Its programming language, called Wolfram, has a powerful matching engine. It uses variadic symbols, individual and sequence variables, and can work modulo A and C theories,

called there flat and orderless theories, respectively. The matching mechanism is explained in tutorials and help files, but to the best of our knowledge, its formal description has never been published. We try to fill this gap, proposing rules which are a further restriction of the above-mentioned strict variant and, in our opinion, describe the input-output behavior and properties of Mathematica's flat and orderless pattern matching. In particular, our analysis suggest that Mathematica's matching corresponds to an incomplete strict variant of a fragment normalized with respect to the given equational theory. The algorithm is not complete. We suppose that incompleteness is a deliberate decision motivated by efficiency reasons.

This paper is an extended and improved version of (Dundua et al., 2019).

Related work. In (Kutsia, 2008), a procedure for flat (A) matching was described and its relation to the correspondent algorithm in Mathematica was discussed. The current work builds on it and extends the results from that paper. Namely, for the variadic A-theory, Kutsia (2008) presented a nonterminating complete procedure and terminating incomplete algorithms, while we propose here a special representation of substitutions, which allows us to design a terminating complete algorithm. The bounded fragment in the current paper generalizes the bounded fragment in (Kutsia, 2008). The strict variant that we consider here has not been studied in (Kutsia, 2008), but the $\mathfrak{F}_{\mathrm{NE}}$ algorithm there uses a similar idea. Moreover, in the current paper we consider also variadic C- and AC-theories. They were not treated in (Kutsia, 2008). Finally, the comparison with Mathematica's behavior is more comprehensive in the current paper than in (Kutsia, 2008). It covers more theories, since in addition to flat, we also discuss matching with the orderless and flat-orderless attributes, as the Mathematica counterparts of variadic commutative and variadic associative-commutative matching. Also, this comparison reflects some changes that, as it turned out, meanwhile happened in the system.

Pattern matching in Mathematica is informally described in (Wolfram, 2003; Trott, 2004) and in other sources on Mathematica programming. We are not aware of any formal treatment of this mechanism and its semantics.

Variadic matching with sequence variables has been used in declarative programming (Dundua, 2014; Dundua et al., 2016; Marin and Kutsia, 2006), reasoning (Kutsia, 2003; Pease and Sutcliffe, 2007), XML processing and transformation (Coelho et al., 2010; Coelho and Florido, 2007; Kutsia and Marin, 2005; Hosoya and Pierce, 2003), program generation (Barthels et al.,

2019; Richardson and Fuchs, 1997; Chasseur and Deville, 1997), rewriting (Hamana, 1997; Buchberger, 1996; Dundua et al., 2009), etc. Corresponding solving methods have been integrated in the mathematical assistant system Theorema (Buchberger et al., 2006), rule-based system PρLog (Dundua et al., 2017) and its predecessor FunLog (Marin and Kutsia, 2003), programming package Sequentica (Marin and Tepeneu, 2003), XML processing language CLP(Flex) (Coelho and Florido, 2004). Variadic matching with regular expression types was studied in (Kutsia and Marin, 2015). Common Logic (CL) (International Organization for Standardization, 2018) is a framework for a family of logic-based languages, designed for knowledge exchange in a heterogeneous network. It comes with variadic symbols and sequence markers (a counterpart of our sequence variables). Syntactic matching for CL was studied in (Kutsia and Marin, 2012).

Recently, a library to extend Python with variadic matching and sequence variables has been developed (Krebber et al., 2017). Pattern matching compiler Tom (Cirstea et al., 2010) supports associative matching. Matching in the combination of order-sorted associative, commutative, and unit element theories is implemented in Maude (Clavel et al., 2007). In Kutsia (2007), it was shown how variadic syntactic matching with sequence variables can be encoded as a special form of order-sorted associative-unit matching. Usefulness of variadic operators and sequence variables in logical frameworks has been discussed in (Horozal et al., 2014; Horozal, 2014).

2. Preliminaries

We assume some familiarity with the standard notions of unification theory (Baader and Snyder, 2001). We consider four pairwise disjoint sets: function symbols $\mathcal{F}$, individual variables $\mathcal{V}_{\text{Ind}}$, sequence variables $\mathcal{V}_{\text{Seq}}$, and function variables $\mathcal{V}_{\text{Fun}}$. All the symbols in $\mathcal{F}$ are *variadic*, i.e., their arity is not fixed. We will use x, y, z for individual variables, $\overline{x}, \overline{y}, \overline{z}$ for sequence variables, X, Y, Z for function variables, and a, b, c, f, g, h for function symbols. The set of variables $\mathcal{V}_{\text{Ind}} \cup \mathcal{V}_{\text{Seq}} \cup \mathcal{V}_{\text{Fun}}$ is denoted by $\mathcal{V}$. *Terms* t and *sequence elements* s are defined by the grammar:

$$t ::= x \mid f(s_1, \ldots, s_n) \mid X(s_1, \ldots, s_n), \quad n \geq 0, \qquad\qquad s ::= t \mid \overline{x}.$$

When it is not ambiguous, we write f for the term $f()$ where $f \in \mathcal{F}$. In particular, we will always write a, b, c for $a(), b(), c()$. Terms are denoted

by t, r and sequence elements by s, q. Finite, possibly empty sequences of terms are denoted by $\tilde{t}$, $\tilde{r}$, while $\tilde{s}$, $\tilde{q}$ are used to denote sequences of sequence elements.

The *set of variables* of a term t is denoted by $\mathcal{V}(t)$. We will use the subscripts Ind, Seq, and Fun to indicate the sets of individual, sequence, and function variables of a term, respectively. A *ground* term is a term without variables. The size of a term t, denoted $size(t)$, is the number of symbols in it. These definitions are generalized for any syntactic object throughout the paper. The *head* of a term is its root symbol. The head of a variable is the variable itself.

A *substitution* is a mapping from individual variables to individual terms, from sequence variables to finite sequences of sequence elements, and from function variables to function symbols or function variables, such that all but finitely many variables are mapped to themselves. (We do not distinguish between a singleton term sequence and its sole element.) We will use lower case Greek letters for substitutions, with ε reserved for the identity substitution.

For a substitution σ, the domain is the set of variables $dom(\sigma) = \{v \in \mathcal{V} \mid \sigma(v) \neq v\}$ and the range is the set of sequences $range(\sigma) = \{\sigma(v) \mid v \in dom(\sigma)\}$. A substitution can be represented explicitly as a function by a finite set of bindings of variables in its domain: $\{v \mapsto \sigma(v) \mid v \in dom(\sigma)\}$. For readability, we put term sequences in parentheses. For instance, the set $\{x \mapsto f(a, \overline{y}), \overline{x} \mapsto (), \overline{y} \mapsto (a, X(f(b)), x), X \mapsto g\}$ is such a representation of the substitution, which maps x to the term $f(a, \overline{y})$, $\overline{x}$ to the empty sequence, $\overline{y}$ to the sequence of three elements $(a, X(f(b)), x)$, and X to g.

Instances of a sequence element s and a sequence $\tilde{s}$ under a substitution σ, denoted, respectively, by $s\sigma$ and $\tilde{s}\sigma$, are defined as follows:

$$x\sigma = \sigma(x), \qquad \overline{x}\sigma = \sigma(\overline{x}), \qquad (f(s_1, \ldots, s_n))\sigma = f(s_1\sigma, \ldots, s_n\sigma),$$
$$(X(s_1, \ldots, s_n))\sigma = \sigma(X)(t_1\sigma, \ldots, t_n\sigma), \quad (s_1, \ldots, s_n)\sigma = (s_1\sigma, \ldots, s_n\sigma).$$

Example 1. Let $\sigma = \{x \mapsto f(a), \overline{x} \mapsto (b, c), \overline{y} \mapsto (), X \mapsto g\}$. Then $(X(x, \overline{x}, f(\overline{y})),\ \overline{x}, \overline{y}, x)\sigma = (g(f(a), b, c, f), b, c, f(a))$. One can also see that nested sequences are not allowed: they are immediately flattened.

Composition of two substitutions σ and ϑ, written $\sigma\vartheta$, is a substitution defined by $(\sigma\vartheta)(v) = \vartheta(\sigma(v))$ for all $v \in \mathcal{V}$.

An *equation* is a pair of individual terms. Given a set E of equations over $\mathcal{F}$ and $\mathcal{V}$, we denote by $\doteq_E$ the least congruence relation on the set of

finite sequences of sequence elements (over $\mathcal{F}$ and $\mathcal{V}$) that is closed under substitution application and contains E. The set $\doteq_E$ is called an *equational theory* defined by E. Slightly abusing the terminology, we will also call the set E an equational theory or an E-theory. When $\tilde{s} \doteq_E \tilde{q}$, we say that $\tilde{s}$ and $\tilde{q}$ *are equal modulo* E. The *signature* of E, denoted $sig(E)$, is the set of all function symbols occurring in E. A function symbol is called *free* with respect to E if it does not occur in $sig(E)$.

A substitution σ is *more general* than a substitution ϑ on a set of variables V modulo an equational theory E, denoted $\sigma \preceq_E^V \vartheta$, if there exists a substitution φ such that $\chi\sigma\varphi \doteq_E \chi\vartheta$ for all individual and sequence variables $\chi \in V$, and $X()\sigma\varphi \doteq_E X()\vartheta$ for all function variables $X \in V$.

Two substitutions σ and ϑ are called *equigeneral* on a set of variables V modulo an equational theory E (E-*equigeneral*) if $\sigma \preceq_E^V \vartheta$ and $\vartheta \preceq_E^V \sigma$.

Solving equations in an equational theory E is called E-*unification*. If one of the sides of an equation is ground, then it is called a *matching equation*, and solving such equations in a theory E is called E-*matching*. We write E-matching equations as $s \ll_E t$, where t is ground. If $E = \emptyset$ (i.e., if all involved function symbols are free), we talk about *syntactic matching*. An E-*matching problem* over $\mathcal{F}$ is a finite set of E-matching equations over $\mathcal{F}$ and $\mathcal{V}$, which is usually denoted by Γ: $\Gamma = \{s_1 \ll_E t_1, \ldots, s_n \ll_E t_n\}$.

An E-*matcher* of Γ is a substitution σ such that $s_i\sigma \doteq_E t_i$ for all $1 \leq i \leq n$. The set of all E-matchers of Γ is denoted by $match_E(\Gamma)$. Γ is E-*matchable*, or E-*solvable*, if $match_E(\Gamma) \neq \emptyset$.

A *complete set of* E-*matchers* of Γ is a set S of substitutions with the following properties:

1. (Correctness) $S \subseteq match_E(\Gamma)$, i.e., each element of S is an E-matcher of Γ;

2. (Completeness) For each $\vartheta \in match_E(\Gamma)$ there exists $\sigma \in S$ with $\sigma \preceq_E^{\mathcal{V}(\Gamma)} \vartheta$.

The set S is a *minimal complete set of* E-*matchers* of Γ with respect to $\mathcal{V}(\Gamma)$ if it is a complete set of E-matchers satisfying the minimality property, which states that two distinct elements of S are incomparable with respect to $\preceq_E^{\mathcal{V}(\Gamma)}$:

3. (minimality) For all $\sigma, \vartheta \in S$, if $\sigma \preceq_E^{\mathcal{V}(\Gamma)} \vartheta$, then $\sigma = \vartheta$.

Now we briefly discuss the *matching type* of equational theories. Its definition is based on the existence and cardinality of the minimal complete set of E-matchers. A *matching problem is of type unitary* (resp. *finitary*, *infinitary*) if its minimal complete set of matchers exists and is a singleton (resp., a finite non-singleton set, an infinite set). It is of *type zero* if the minimal complete set of matchers does not exist. These types are denoted respectively by $1, \omega, \infty, 0$, and are ordered as $1 < \omega < \infty < 0$. An *equational theory E has matching of type unitary* (resp. *finitary*, *infinitary*, *zero*) if the $<$-maximal type of E-matching problems in this theory is 1 (resp. ω, ∞, 0). See (Baader and Snyder, 2001) for more information about unification/matching types.

Below, in termination proofs, we will need a well-founded ordering on multisets of natural numbers. For this, the Dershowitz-Manna ordering on multisets (Dershowitz and Manna, 1979) will be used, which we denote by $>_{\mathrm{DM}}$. It is defined as follows:

Definition 1 (Multiset Ordering $>_{\mathrm{DM}}$)**.** Let S be a partially ordered set by an ordering $>$, and $\mathcal{M}(S)$ be the set of all finite multisets with elements taken from S. Then for $M, M' \in \mathcal{M}(S)$, we say that $M >_{\mathrm{DM}} M'$ if for some multisets $X, Y \in \mathcal{M}(S)$ where $\emptyset \neq X \subseteq M$, we have $M' = (M \setminus X) \cup Y$ and for all $y \in Y$ there exists $x \in X$ such that $x > y$.

Dershowitz and Manna (1979) explain this definition also in words: "a multiset is reduced by the removal of at least one element (those in X) and their replacement with any finite number – possibly zero – of elements (those in Y), each of which is smaller than one of the elements that have been removed."

In our proofs, we will have $\mathbb{N}$ in the role of S with the standard ordering on natural numbers.

3. Variadic Equational Matching Problems

In this paper we consider equational theories that specify pretty common properties of variadic function symbols: counterparts of *associativity* and *commutativity*. They are defined by the axioms below (for a function symbol f):

$f(\overline{x}, f(\overline{y}), \overline{z}) \doteq f(\overline{x}, \overline{y}, \overline{z})$	variadic associativity for f, $\mathsf{A}(f)$
$f(\overline{x}, x, \overline{y}, y, \overline{z}) \doteq f(\overline{x}, y, \overline{y}, x, \overline{z})$	variadic commutativity for f, $\mathsf{C}(f)$

The $\mathsf{A}(f)$ axiom asserts that nested occurrences of f can be flattened out. The $\mathsf{C}(f)$ axiom says that the order of arguments of f does not matter. Below we often omit the word "variadic" and write associativity and commutativity instead of variadic associativity and variadic commutativity. We also say f is A, C, or AC if, respectively, only $\mathsf{A}(f)$, only $\mathsf{C}(f)$, or both $\mathsf{A}(f)$ and $\mathsf{C}(f)$ hold for f.

Example 2. We can illustrate the notions of equality modulo $\mathsf{A}(f)$ and/or $\mathsf{C}(f)$ for some f with the following examples:

- $f(f(f(),a)) \doteq_{\mathsf{A}(f)} f(f(a)) \doteq_{\mathsf{A}(f)} f(f(),a) \doteq_{\mathsf{A}(f)} f(a)$.
- $(f(f()),f(),f(a)) \doteq_{\mathsf{A}(f)} (f(),f(),f(a))$.
- $(f(f()),f(),f(a)) \not\doteq_{\mathsf{A}(f)} f(a)$.
- $f(f(),\ldots,f()) \doteq_{\mathsf{A}(f)} f()$, but $(f(),\ldots,f()) \not\doteq_{\mathsf{A}(f)} f()$.
- $f(f(f())) \doteq_{\mathsf{A}(f)} f(f()) \doteq_{\mathsf{A}(f)} f()$.
- $f(f(a),b,f(f(),c))) \doteq_{\mathsf{A}(f)} f(a,b,c)$.
- $f(f(a),b,f(f(),c))) \doteq_{\mathsf{C}(f)} f(b,f(f(),c),f(a))$.
- $f(f(a),b,f(f(),c))) \doteq_{\mathsf{A}(f),\mathsf{C}(f)} f(b,f(c),a) \doteq_{\mathsf{A}(f),\mathsf{C}(f)} f(b,c,a)$.

An *associative normal form* (A-*normal form*) of a term or a sequence is obtained by rewriting it with the associativity axiom from left to right as long as possible. In Example 2, for instance, $f(a)$ is the A-normal form of the terms $f(f(f(),a))$, $f(f(a))$, $f(f(),a)$, $f(f(),a)$, and $f(a)$; $(f(),f(),f(a))$ is the normal form of the sequences $(f(f()),f(),f(a))$, $(f(),f(),f(a))$, and $(f(f()),f(),f(a))$; $f(a,b,c)$ is the A-normal form of $f(f(a),b,f(f(),c))$. A term or a sequence is in A-normal form if it is its own A-normal form

We introduce a strict total order on function symbols and extend it to ground terms and term sequences so that the obtained ordering is also total. A *commutative normal form* (C-*normal form*) of a ground term is obtained by rearranging arguments of commutative function symbols to obtain the minimal term with respect to the defined ordering. An *associative-commutative normal form* (AC-*normal form*) of a ground term t is the C-normal form of the A-normal form of t.

A C-normal form (resp. an AC-normal form) of a sequence of ground terms $(t_1, \ldots, t_n)$ is a sequence of ground terms $(t'_1, \ldots, t'_n)$ where t'_i is a C-normal form (resp. AC-normal form) of t_i for all $1 \leq i \leq n$.

Similarly, a C-normal form (resp. an AC-normal form) of a multiset of ground terms $\{\!\{t_1, \ldots, t_n\}\!\}$ is a multiset of ground terms $\{\!\{t'_1, \ldots, t'_n\}\!\}$ where t'_i is a C-normal form (resp. AC-normal form) of t_i for all $1 \leq i \leq n$.

A ground term, sequence of ground terms, or a multiset of ground terms is *in* C- (resp. AC-) *normal form* if it is its own C- (resp. AC-) normal form. All these normal forms are unique.

The notion of normal form extends to substitutions straightforwardly: σ is in A- (C-, AC-) *normal form* if $\chi\sigma$ is in A- (C-, AC-) normal form for all individual and sequence variables χ.

In A- and AC-theories there exist matching problems whose minimal complete set of matchers is infinite. This is related to the flatness property of variadic associative symbols, and originates from flat matching (Kutsia, 2002a, 2008). The simplest such problem is $f(\overline{x}) \ll_E f()$, where $\mathsf{A}(f) \in E$. Its complete solution set is $\{\{\overline{x} \mapsto ()\}, \{\overline{x} \mapsto f()\}, \{\overline{x} \mapsto (f(), f())\}, \ldots\}$, which is based on the fact that $f(f(), \ldots, f()) \doteq_E f()$ when $\mathsf{A}(f) \in E$. It, naturally, implies that any matching procedure that directly enumerates a complete set of A- or AC-matchers is non-terminating.

In general, our matching problems are formulated in a theory that may contain several A, C, or AC-symbols.

Below we design variadic A-,C-, and AC-matching algorithms first for a special case when no variable occurs more than once in matching problems. Such matching problems are called *linear*. The general, nonlinear case will be considered after that. To solve a nonlinear problem, we first linearize it, solve the obtained linear problem by the corresponding algorithm, and then combine the computed solutions into matchers of the original problem. In the next two sections we discuss these algorithms in detail.

4. Matching algorithm for linear problems

We formulate our matching algorithm in a rule-based manner for linear problems. The rules operate on a matching equation and return a set of matching equations. They also produce a set of equations, which we call a *solved set*, denoted in the rules by S.

Before continuing with the rules, we need to define notions related to solved sets.

Definition 2 (Solved Set of Equations)**.** *Solved sets* are sets of equations, whose elements are called *solved equations* and have one of six possible forms: $x \approx t$, $X \approx g$, $\overline{x} \approx \tilde{r}$, $\overline{x} \approx \{\!\{\tilde{r}\}\!\}$, $\overline{x} \approx \tilde{t}[f]$, and $\overline{x} \approx \{\!\{\tilde{t}\}\!\}[f]$, where the symbol $\{\!\{\cdot\}\!\}$ denotes multisets. In equations $\overline{x} \approx \tilde{t}[f]$ and $\overline{x} \approx \{\!\{\tilde{t}\}\!\}[f]$, f is associative or associative-commutative and no term in $\tilde{t}$ has f as its head. Besides, each variable that appears in a solved set, appears there only once, in the left hand side of an equation. The right hand sides are ground and in (A-, C-, AC-) normal form.

Definition 3 (Set of Substitutions Generated by a Solved Equation)**.** Each solved equation eq generates a set of substitutions, denoted by $\Sigma(\mathsf{eq})$, as follows:

$$
\begin{aligned}
&\Sigma(x \approx t) = \{\{x \mapsto t\}\}.\\
&\Sigma(X \approx f) = \{\{X \mapsto f\}\}.\\
&\Sigma(\overline{x} \approx \tilde{t}) = \{\{\overline{x} \mapsto \tilde{t}\}\}.\\
&\Sigma(\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}) =\\
&\qquad \{\{\overline{x} \mapsto (t_{\pi(1)}, \ldots, t_{\pi(n)})\} \mid \pi \text{ is a permutation of } (1, \ldots, n)\}.\\
&\Sigma(\overline{x} \approx (t_1, \ldots, t_n)[f]) =\\
&\qquad \{\{\overline{x} \mapsto \tilde{r}\} \mid f(t_1, \ldots, t_n) \doteq_{\mathsf{A}(f)} f(\tilde{r}) \text{ and } \tilde{r} \text{ is in A-normal form}\},\\
&\qquad \text{where } n \geq 0.\\
&\Sigma(\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}[f]) =\\
&\qquad \{\{\overline{x} \mapsto \tilde{r}\} \mid f(t_{\pi(1)}, \ldots, t_{\pi(n)}) \doteq_{\mathsf{A}(f)} f(\tilde{r}), \text{ where } \pi \text{ is a permutation}\\
&\qquad\qquad \text{of } (1, \ldots, n) \text{ and } \tilde{r} \text{ is in AC-normal form}\},\\
&\qquad \text{where } n \geq 0.
\end{aligned}
$$

To explain in words, each element of $\Sigma(\overline{x} \approx (t_1, \ldots, t_n)[f])$ is a substitution that maps $\overline{x}$ either to $(t_1, \ldots, t_n)$ or to a sequence obtained from $(t_1, \ldots, t_n)$ by applying f to some of its subsequence(s). That can be, in particular, the empty subsequence, i.e., $f()$'s may get inserted. Similarly, each element of $\Sigma(\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}[f])$ is a substitution that maps $\overline{x}$ to a permutation of $(t_1, \ldots, t_n)$ or to a sequence obtained from a permutation of $(t_1, \ldots, t_n)$ by applying f to some of its subsequence(s) (which can be also empty). The AC-normal form requirement in the definition guarantees that if f is applied to a subsequence $\tilde{t}$ of a permutation $(t_1, \ldots, t_n)$, then $f(\tilde{t})$ is AC-normalized.

Example 3. Each of the last two cases in Definition 3 actually defines an infinite set of pairwise $\leq_E^{\{\overline{x}\}}$-incomparable substitutions. For instance, $\Sigma(\overline{x} \approx (t_1, t_2, t_3)[f])$ is the set

$$\begin{aligned}
&\{\{\overline{x} \mapsto (\tilde{r}_1, r_1, \tilde{r}_2, r_2, \tilde{r}_3, r_3, \tilde{r}_4)\} \\
&\qquad \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 4,\ r_j \in \{t_j, f(t_j)\},\ 1 \leq j \leq 3\} \\
&\cup \{\{\overline{x} \mapsto (\tilde{r}_1, f(t_1, t_2), \tilde{r}_2, r_3, \tilde{r}_3)\} \\
&\qquad \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 3,\ r_3 \in \{t_3, f(t_3)\}\} \\
&\cup \{\{\overline{x} \mapsto (\tilde{r}_1, r_1, \tilde{r}_2, f(t_2, t_3), \tilde{r}_3)\} \\
&\qquad \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 3,\ r_1 \in \{t_1, f(t_1)\}\} \\
&\cup \{\{\overline{x} \mapsto (\tilde{r}_1, f(t_1, t_2, t_3), \tilde{r}_2)\} \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 2\}.
\end{aligned}$$

Here * is the Kleene star: $\{f()\}^*$ is the set of sequences $\{(), f(), (f(), f()), (f(), f(), f()), \ldots\}$.

Assume that the function symbol names are ordered alphabetically (for C-ordering). Then $\Sigma(\overline{x} \approx \{\!\{a, b\}\!\}[f])$ is the set

$$\begin{aligned}
&\{\{\overline{x} \mapsto (\tilde{r}_1, r_1, \tilde{r}_2, r_2, \tilde{r}_3)\} \\
&\qquad \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 3,\ r_1 \in \{a, f(a)\},\ r_2 \in \{b, f(b)\}\} \\
&\cup \{\{\overline{x} \mapsto (\tilde{r}_1, r_1, \tilde{r}_2, r_2, \tilde{r}_3)\} \\
&\qquad \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 3,\ r_1 \in \{b, f(b)\},\ r_2 \in \{a, f(a)\}\} \\
&\cup \{\{\overline{x} \mapsto (\tilde{r}_1, f(a, b), \tilde{r}_2)\} \mid \tilde{r}_i \in \{f()\}^*,\ 1 \leq i \leq 3\}.
\end{aligned}$$

Note that $f(b, a)$ does not appear in the range of the substitutions due to the AC-normal form requirement in the definition of $\Sigma(\overline{x} \approx \{\!\{a, b\}\!\}[f])$: the AC-normal form of $f(b, a)$ is $f(a, b)$ (with respect to the given ordering on function symbols).

Definition 4 (Set of Substitutions Generated by Solved Sets)**.** We extend Σ to solved sets:

$$\begin{aligned}
&\Sigma(\emptyset) = \{\varepsilon\}. \\
&\Sigma(\{\mathsf{eq}_1, \ldots, \mathsf{eq}_n\}) = \{\sigma_1 \cup \cdots \cup \sigma_n \mid \sigma_i \in \Sigma(\mathsf{eq}_i),\ 1 \leq i \leq n\}, \\
&\qquad \text{where } n \geq 1 \text{ and } \{\mathsf{eq}_1, \ldots, \mathsf{eq}_n\} \text{ is a solved set.}
\end{aligned}$$

If $\mathbf{S}$ is a set of solved sets, then by $\boldsymbol{\Sigma}(\mathbf{S})$ we denote the set

$$\boldsymbol{\Sigma}(\mathbf{S}) := \bigcup_{S \in \mathbf{S}} \Sigma(S).$$

The matching algorithm defined below will be based on applications of transformation rules. It is important to note that terms in the left hand sides of the rules are kept in normal forms with respect to associativity, and those in the right hand side (ground terms) are kept in normal forms with respect to associativity and commutativity. The transformation rules are divided into three groups: the common rules, rules for associative symbols, and the rules that deal with commutativity.

The rules have the form $s \ll_E t \rightsquigarrow_S \Gamma$, where $s \ll_E t$ is an E-matching equation, S is a solved set (which is either empty or consists of a single solved equation), and Γ is a finite set of E-matching equations. Intuitively, such a rule transforms $s \ll_E t$ into Γ, and also records information in S how a particular variable is supposed to be instantiated in this transformation.

Common Rules. The common rules apply in any theory.

T: **Trivial**

$s \ll_E s \rightsquigarrow_\emptyset \emptyset$.

IVE: **Individual variable elimination**

$x \ll_E t \rightsquigarrow_S \emptyset, \qquad$ where $S = \{x \approx t\}$.

FVE: **Function variable elimination**

$X(\tilde{s}) \ll_E f(\tilde{t}) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t})\}, \qquad$ where $S = \{X \approx f\}$.

Rules for free symbols. These rules apply when f is free.

Dec-F: **Decomposition under free head**

$f(s, \tilde{s}) \ll_E f(t, \tilde{t}) \rightsquigarrow_\emptyset \{s \ll_E t,\ f(\tilde{s}) \ll_E f(\tilde{t})\}$,

where f is free and $s \notin \mathcal{V}_{\mathrm{Seq}}$.

SVE-F: **Sequence variable elimination under free head**

$f(\overline{x}, \tilde{s}) \ll_E f(\tilde{t}_1, \tilde{t}_2) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_2)\}$,

where f is free and $S = \{\overline{x} \approx \tilde{t}_1\}$.

Rules for commutative symbols. These rules apply when f is commutative but not associative.

Dec-C: **Decomposition under commutative head**

$f(s, \tilde{s}) \ll_E f(\tilde{t}_1, t, \tilde{t}_2) \rightsquigarrow_\emptyset \{s \ll_E t,\ f(\tilde{s}) \ll_E f(\tilde{t}_1, \tilde{t}_2)\}$,

where f is commutative and non-associative and $s \notin \mathcal{V}_{\mathrm{Seq}}$.

SVE-C: **Sequence variable elimination under commutative head**

$f(\overline{x}, \tilde{s}) \ll_E f(\tilde{t}_1, t_1, \tilde{t}_2, \ldots, \tilde{t}_n, t_n, \tilde{t}_{n+1}) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_1, \ldots, \tilde{t}_{n+1})\}$,

where $n \geq 0$, f is commutative and non-associative, $S = \{\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}\}$.

Rules for associative symbols. These rules apply when f is associative but not commutative.

Dec-A: **Decomposition under associative head**

$f(s, \tilde{s}) \ll_E f(t, \tilde{t}) \rightsquigarrow_\emptyset \{s \ll_E t,\ f(\tilde{s}) \ll_E f(\tilde{t})\}$,

where f is associative and non-commutative and $s \notin \mathcal{V}_{\mathrm{Seq}}$.

SVE-A: **Sequence variable elimination under associative head**

$f(\overline{x}, \tilde{s}) \ll_E f(\tilde{t}_1, \tilde{t}_2) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_2)\}$,

where f is associative and non-commutative and $S = \{\overline{x} \approx (\tilde{t}_1)[f]\}$.

FVE-A: **Function variable elimination under associative head**

$f(X(\tilde{s}_1), \tilde{s}_2) \ll_E f(\tilde{t}) \rightsquigarrow_S \{f(\tilde{s}_1, \tilde{s}_2) \ll_E f(\tilde{t})\}$,

where f is associative and non-commutative and $S = \{X \approx f\}$.

IVE-A: **Individual variable elimination under associative head**

$f(x, \tilde{s}) \ll_E f(\tilde{t}_1, \tilde{t}_2) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_2)\}$,

where f is associative and non-commutative and $S = \{x \approx f(\tilde{t}_1)\}$.

Rules for associative-commutative symbols. These rules apply when f is both associative and commutative.

Dec-AC: **Decomposition under AC head**

$f(s, \tilde{s}) \ll_E f(\tilde{t}_1, t, \tilde{t}_2) \rightsquigarrow_\emptyset \{s \ll_E t,\ f(\tilde{s}) \ll_E f(\tilde{t}_1, \tilde{t}_2)\}$,

where f is associative-commutative and $s \notin \mathcal{V}_{\mathrm{Seq}}$.

SVE-AC: **Sequence variable elimination under AC head**

$f(\overline{x}, \tilde{s}) \ll_E f(\tilde{t}_1, t_1, \tilde{t}_2, \ldots, \tilde{t}_n, t_n, \tilde{t}_{n+1}) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_1, \ldots, \tilde{t}_{n+1})\}$,

where $n \geq 0$, f is associative-commutative and $S = \{\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}[f]\}$.

FVE-AC: **Function variable elimination under AC head**

$f(X(\tilde{s}_1), \tilde{s}_2) \ll_E f(\tilde{t}) \rightsquigarrow_S \{f(\tilde{s}_1, \tilde{s}_2) \ll_E f(\tilde{t})\}$,

where f is associative-commutative and $S = \{X \approx f\}$.

IVE-AC: **Individual variable elimination under AC head**

$f(x, \tilde{s}) \ll_E f(\tilde{t}_1, t_1, \tilde{t}_2, \ldots, \tilde{t}_n, t_n, \tilde{t}_{n+1}) \rightsquigarrow_S \{f(\tilde{s}) \ll_E f(\tilde{t}_1, \ldots, \tilde{t}_{n+1})\}$,
where $n \geq 0$, f is associative-commutative and $S = \{x \approx f(t_1, \ldots, t_n)\}$.

The matching algorithm. The form and the conditions of the rules guarantee that no two rules apply to the same equation except if one of them is Dec-A or Dec-AC. Dec-A can apply to the equation that is transformed by FVE-A or IVE-A. Similarly, Dec-AC is an alternative of FVE-AC or IVE-AC.[2] Moreover, it is possible that a rule transforms the same equation in multiple ways. All sequence variable elimination and individual variable elimination rules are like that, depending on the choice of the subsequence of the right hand side they match to. Also, Dec-C and Dec-AC rules may choose the t from the right hand side in different ways.

The matching algorithm **LM** ("linear matching" to indicate that it works on linear matching problems) works on pairs $\Gamma; S$, where Γ is a linear matching problem consisting of equations in A-normalized left hand sides and AC-normalized right hand sides. It selects an equation from Γ and transforms it by the applicable rule, if such a rule exists. Assume a rule R transforms $s \ll_E t \rightsquigarrow_S \Gamma$. One step of **LM** is then performed as $\{s \ll t\} \uplus \Gamma'; S' \rightsquigarrow_{\mathsf{R}} \Gamma \cup \Gamma'; S \cup S'$, where $\uplus$ is the disjoint union symbol. A sequence of rule applications is called a *derivation*. If the chosen rule can be applied in multiple ways, one alternative is chosen nondeterministically. Such a nondeterminism introduces branching in the derivation tree.

Given a matching problem Γ, we create the initial system $\Gamma; \emptyset$ and start applying the rules. If a derivation reaches a terminal system $\emptyset; S$, then it is called a *successful derivation* and $\emptyset; S$ is called a *success leaf*. If a derivation cannot be extended further from a system of the form $\Gamma'; S'$ where $\Gamma' \neq \emptyset$, then it is called *failed*.

The set S at the success leaf $\emptyset; S$ of a derivation $\Gamma; \emptyset \rightsquigarrow^* \emptyset; S$ is called an *answer of* Γ *computed by* **LM**, or just a *computed answer* of Γ. The set of answers of Γ computed by the algorithm **LM** is denoted by $\mathbf{LM}(\Gamma)$.

Example 4. Some linear matching problems and the corresponding computed answers:

[2] In the published version, these properties of Dec-A or Dec-AC are overlooked to be mentioned at this place, although the rules are correctly applied throughout the paper.

- $\Gamma = \{f(\overline{x}) \ll_E f(a,b)\}$, $E = \{\mathsf{C}(f)\}$, $\mathbf{LM}(\Gamma) = \{\{\overline{x} \approx \{\!\{a,b\}\!\}\}\}$.
- $\Gamma = \{f(\overline{x}) \ll_E f(a,b)\}$, $E = \{\mathsf{A}(f)\}$, $\mathbf{LM}(\Gamma) = \{\{\overline{x} \approx (a,b)[f]\}\}$.
- $\Gamma = \{f(\overline{x}) \ll_E f(a,b)\}$, $E = \{\mathsf{A}(f), \mathsf{C}(f)\}$, $\mathbf{LM}(\Gamma) = \{\{\overline{x} \approx \{\!\{a,b\}\!\}[f]\}$.
- $\Gamma = \{f(x,\overline{y}) \ll_E f(a,b)\}$, $E = \{\mathsf{A}(f), \mathsf{C}(f)\}$, $\mathbf{LM}(\Gamma) = \{\{x \approx a, \overline{y} \approx \{\!\{b\}\!\}[f]\}, \{x \approx f(a), \overline{y} \approx \{\!\{b\}\!\}[f]\}, \{x \approx b, \overline{y} \approx \{\!\{a\}\!\}[f]\}, \{x \approx f(b), \overline{y} \approx \{\!\{a\}\!\}[f]\}, \{x \approx f(), \overline{y} \approx \{\!\{a,b\}\!\}[f]\}, \{x \approx f(a,b), \overline{y} \approx \emptyset[f]\}\}$.
- $\Gamma = \{f(X(y),b,z) \ll_E f(a,b,b)\}$, $E = \{\mathsf{A}(f)\}$, $\mathbf{LM}(\Gamma) = \{\{X \approx f, y \approx a, z \approx b\}, \{X \approx f, y \approx a, z \approx f(b)\}, \{X \approx f, y \approx f(a), z \approx b\}, \{X \approx f, y \approx f(a), z \approx f(b)\}, \{X \approx f, y \approx f(a,b), z \approx f()\}\}$.

Recall that our matching problems are formulated in a theory that may contain one or more A, C, or AC-symbols. In the theorems proven below, the equational theory E refers to such a theory. The considered matching problems are linear.

Theorem 1 (Computed solved set)**.** *For a linear E-matching problem Γ, every element of $\mathbf{LM}(\Gamma)$ is a solved set.*

Proof. Let $S \in \mathbf{LM}(\Gamma)$. Since Γ is linear, by inspection of the rules it is clear that no variable will occur more than once in S. Moreover, the right hand sides of equations in S are E-normalized. Also, in the equations of the form $x \approx \{\!\{\tilde{t}\}\!\}[f]$ and $x \approx \tilde{t}[f]$, f is not the head of any of the terms in $\tilde{t}$. Hence, S is a solved set. □

Theorem 2 (Termination of **LM**)**.** *The algorithm* **LM** *terminates for every input.*

Proof. Let the complexity measure of a linear E-matching problem Γ be the pair (N, M), where N is the number of variables in Γ, and M is the multiset of the sizes of the right hand sides of equations in Γ. Measures are compared lexicographically. The obtained ordering is well-founded, as a lexicographic combination of two well-founded orderings: $>$ on natural numbers and $>_{\mathrm{DM}}$ on multisets of natural numbers. The rules strictly reduce this measure. It is easy to see since each variable elimination rule reduces N and the other rules reduce M without changing N. □

For the soundness theorem, we will need the following lemma:

Lemma 1. *If* $\Gamma_1; S_1 \rightsquigarrow_{\mathsf{R}} \Gamma_2; S_1 \cup S$ *is a step in* **LM**, *then* $match_E(\Gamma_2) = match_E(\Gamma_1\sigma)$ *for each* $\sigma \in \Sigma(S)$.

Proof. We prove the lemma only for the rule SVE-AC. For the other rules the proof is simpler.

If the step is performed by SVE-AC, then for some Γ and an AC symbol f we have

$$\begin{aligned}\Gamma_1 &= \{f(\overline{x}, \tilde{s}) \ll_E f(\tilde{t}_1, t_1, \ldots, t_n, \tilde{t}_n)\} \cup \Gamma,\\ \Gamma_2 &= \{f(\tilde{s}) \ll_E f(\tilde{t}_1, \ldots, \tilde{t}_{n+1})\} \cup \Gamma,\\ S &= \{\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}[f]\}.\end{aligned}$$

Let $\sigma \in \Sigma(S)$. Then for $\varphi \in match_E(\Gamma_2)$ we have

$$f(\tilde{s})\varphi = f(\tilde{s}\varphi) \approx_{\mathsf{AC}} f(\tilde{t}_1, \ldots, \tilde{t}_{n+1}). \tag{1}$$

Then $dom(s) = \{\overline{x}\}$. Since we work with linear problems, $\overline{x} \notin \mathcal{V}(f(\tilde{s}))$. Therefore from (1) we get

$$\begin{aligned} f(\overline{x}, \tilde{s})\sigma\varphi \approx_{\mathsf{A}} \;& f(t_1, \ldots, t_n, \tilde{s}\sigma\varphi) \approx_{\mathsf{AC}} f(t_1, \ldots, t_n, \tilde{s}\varphi)\\ \approx_{\mathsf{AC}} \;& f(t_1, \ldots, t_n, \tilde{t}_1, \ldots, \tilde{t}_n) \approx_{\mathsf{AC}} f(\tilde{t}_1, t_1, \ldots, t_n, \tilde{t}_n).\end{aligned} \tag{2}$$

Since Γ remains unchanged during the rule application, from (2) we get that $\varphi \in match_E(\Gamma_1\sigma)$. This proves $match_E(\Gamma_2) \subseteq match_E(\Gamma_1\sigma)$. The other direction can be proved analogously. □

Corollary 1. *For each* n, *if* $\Gamma; \emptyset \rightsquigarrow^n \emptyset; S$ *is a sequence of rule applications in* **LM**, *then* $\Sigma(S) \subseteq match_E(\Gamma)$, *where* Γ *is linear.*

Proof. By induction on n, using Lemma 1 to make the step. □

Theorem 3 (Soundness)**.** $\mathbf{\Sigma}(\mathbf{LM}(\Gamma)) \subseteq match_E(\Gamma)$ *for any linear E-matching problem* Γ.

Proof. By Corollary 1, for each $S \in \mathbf{LM}(\Gamma)$ we have $\Sigma(S) \subseteq match_E(\Gamma)$. By the definition, $\mathbf{\Sigma}(\mathbf{LM}(\Gamma)) = \cup_{S \in \mathbf{LM}(\Gamma)} \Sigma(S)$ and, hence, we get $\mathbf{\Sigma}(\mathbf{LM}(\Gamma)) \subseteq match_E(\Gamma)$. □

Theorem 4 (Completeness)**.** *Let* Γ *be a linear E-matching problem. Assume* $\sigma \in match_E(\Gamma)$ *and it is in E-normal form. Then* $\sigma|_{\mathcal{V}(\Gamma)} \in \mathbf{\Sigma}(\mathbf{LM}(\Gamma))$.

Proof. We prove the theorem by constructing the derivation that starts from $\Gamma; \emptyset$ and ends with $\emptyset; S$ such that $\sigma|_{\mathcal{V}(\Gamma)} \in \Sigma(S)$.

Assume that we have already constructed $\Gamma; \emptyset \rightsquigarrow^n \Gamma_n; S_n$ such that $\sigma \in match_E(\Gamma_n)$ and $\sigma|_{\mathcal{V}(S_n)} \in \Sigma(S_n)$. We show that we can make the next step $\Gamma_n; S_n \rightsquigarrow \Gamma_{n+1}; S_{n+1}$ such that

$$\sigma \in match_E(\Gamma_{n+1}), \text{ and} \tag{3}$$

$$\sigma|_{\mathcal{V}(S_{n+1})} \in \Sigma(S_{n+1}). \tag{4}$$

Let us select an equation $s \ll_E t \in \Gamma_n$, assume $\Gamma_n = \{s \ll_E t\} \uplus \Gamma'$, and show how to make the inference step. If $head(s) \neq head(t)$, then $head(s)$ should be a variable. Otherwise Γ_n would not be solvable, which contradicts the assumption that σ is a matcher of Γ_n.

If s is a variable x, then we have $\sigma(x) = t$. we extend the derivation by the IVE rule, which gives $\Gamma_{n+1} = \Gamma'$ and $S_{n+1} = S_n \cup \{x \approx t\}$. Then $\sigma \in match_E(\Gamma_{n+1})$, $\sigma|_{\mathcal{V}(S_{n+1})} = \sigma|_{\mathcal{V}(S_n)} \cup \{x \mapsto t\}$, and $\sigma|_{\mathcal{V}(S_{n+1})} \in \Sigma(S_{n+1})$.

In a similar manner, we can make the step when s has a form $X(\tilde{s})$.

Now assume $head(s) = head(t)$. If s and t are the same, then the step is made by the T rule and the conditions (3) and (4) are satisfied.

Assume now $s = f(s_0, s_1, \ldots, s_n)$ and f is free. If $s_0 \notin \mathcal{V}_{\text{Seq}}$, we construct the step by the Dec-F rule. Γ_n and Γ_{n+1} have the same set of matchers, $S_n = S_{n+1}$ and the conditions (3) and (4) hold. If $s_0 = \overline{x}$, then t should have the form $f(\tilde{t}_1, \tilde{t}_2)$, where $\sigma(\overline{x}) = \tilde{t}_1$. We make the step by the SVE-F rule, choosing exactly $\tilde{t}_1$. Then $\Gamma_{n+1} = \{f(s_1, \ldots, s_n) \ll_E f(\tilde{t}_2)\} \cup \Gamma'$ and $S_{n+1} = S_n \cup \{\overline{x} \approx \tilde{t}_1\}$. Again, it is obvious that (3) and (4) hold.

The reasoning is similar when $head(s)$ is commutative. Now we consider the case when it is associative-commutative and s_0 is a sequence variable. The other remaining cases are similar or simpler. We have $s = f(\overline{x}, s_1, \ldots, s_n)$ and f is AC. Let $\tilde{r} = \sigma(\overline{x})$. Let $\tilde{t}_1$ be obtained from $\tilde{r}$ by normalization and removing all occurrences of $f()$ as elements of $\tilde{r}$. Then $t \approx_{\mathsf{AC}} f(\tilde{t}_1, \tilde{t}_2)$. We make the step with SVE-AC, choosing $t_1, \ldots, t_n$ from the arguments of t so that $\{\!\{t_1, \ldots, t_n\}\!\} = \{\!\{\tilde{t}_1\}\!\}$. Then $\Gamma_{n+1} = \{f(s_1, \ldots, s_n) \ll_E f(\tilde{t}_2)\} \uplus \Gamma'$ and the condition 3 holds. As for S_{n+1}, we have $S_{n+1} = S_n \cup \{\overline{x} \approx \{\!\{\tilde{t}_1\}\!\}[f]\}$, and by the definition of Σ, the condition 4 is also satisfied.

By iterating this process, we obtain a final state $\emptyset; S$ such that $\sigma|_{\mathcal{V}(S)} \in \Sigma(S)$. But $\mathcal{V}(S) = \mathcal{V}(\Gamma)$, which finishes the proof. □

Theorem 5 (Minimality)**.** $\mathbf{\Sigma(LM}(\Gamma))$ *is minimal for any linear E-matching problem* Γ.

Proof. First, note that the rules in $\mathbf{LM}(\Gamma)$ do not introduce any fresh variable, and every variable from $\mathcal{V}(\Gamma)$ is eventually moved to the solved set (unless the algorithm fails and there is no solution). It implies that all substitutions in $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ have the same domain, which is $\mathcal{V}(\Gamma)$.

Assume now by contradiction that there exists $\sigma, \vartheta \in \mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ such that $\sigma \leq_E^{\mathcal{V}(\Gamma)} \vartheta$ and $\sigma \neq \vartheta$. Since $dom(\sigma) = dom(\vartheta) = \mathcal{V}(\Gamma)$, we have that $\sigma(v)$ and $\vartheta(v)$ are ground for any $v \in \mathcal{V}(\Gamma)$. But then $\sigma \leq_E^{\mathcal{V}(\Gamma)} \vartheta$ implies also $\vartheta \leq_E^{\mathcal{V}(\Gamma)} \sigma$, i.e., σ and ϑ are E-equigeneral (on $\mathcal{V}(\Gamma)$). Since we kept the right hand sides E-normalized, the transformation rules guarantee that the terms in each computed solved form in $\mathbf{LM}(\Gamma)$ are in E-normal form. From this fact and the definition of Σ, the ranges of substitutions in $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ are E-normalized. Therefore, since the normal forms for our equational theories are unique, equigenerality of $\sigma(v)$ and $\vartheta(v)$ imply that $\sigma(v) = \vartheta(v)$ for any $v \in \mathcal{V}(\Gamma)$. Together with $dom(\sigma) = dom(\vartheta)$, it implies that $\sigma = \vartheta$, which is a contradiction. Hence, $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ is minimal. □

Example 5. Note that if we did not require normalization of the right hand sides in matching problems, $\mathbf{LM}(\Gamma)$ may contain elements that are not solved sets. This can be illustrated, e.g., by the solutions to the E-matching problem $\Gamma = \{f(x, y) \ll_E f(f(a, b), f(b, a))\}$, where $\{\mathsf{C}(f)\} = E$. In this case, $\mathbf{LM}(\Gamma)$ would be $\{\{x \approx f(a, b),\ y \approx f(b, a)\}, \{x \approx f(b, a),\ y \approx f(a, b)\}\}$, and its elements are not solved sets, since the right hand sides are not normalized.

Moreover, it would have the consequence that $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ would not be minimal, since we would get $\mathbf{\Sigma}(\mathbf{LM}(\Gamma)) = \{\sigma_1, \sigma_2\}$, where $\sigma_1 = \{x \mapsto f(a, b),\ y \mapsto f(b, a)\}$ and $\sigma_2 = \{x \mapsto f(b, a),\ y \mapsto f(a, b)\}$ with $\sigma_1 \simeq_{\mathsf{C}(f)} \sigma_2$.

With the normalized right hand sides, $\mathbf{LM}(\Gamma) = \{\{x \approx f(a, b),\ y \approx f(a, b)\}\}$ and $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ contains only the substitution $\{x \mapsto f(a, b),\ y \mapsto f(a, b)\}$, although it is still computed twice.

Example 6. For minimality of $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$, normalization of $\overline{x} \mapsto \tilde{r}$ in the definitions of $\Sigma(\overline{x} \approx (t_1, \ldots, t_n)[f])$ and $\Sigma(\overline{x} \approx \{\!\{t_1, \ldots, t_n\}\!\}[f])$ (Definition 3) is important. For instance, without this requirement, for $\Gamma = \{f(\overline{x}) \ll_E f(a, b)\}$ where $E = \{\mathsf{A}(f), \mathsf{C}(f)\}$, the algorithm **LM** gives the solved set $\{\!\{a, b\}\!\}[f]$, and $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ contains the substitutions $\{\overline{x} \mapsto f(a, b)\}$ and $\{\overline{x} \mapsto f(b, a)\}$, among others.

Theorem 6 (Types of variadic linear C-, A-, and AC-matching problems)**.**

- *The type of variadic linear syntactic and* C*-matching problems is at most finitary.*
- *The type of variadic linear* A*- and* AC*-matching problems is at most infinitary.*

Proof. By Theorem 5, every linear variadic E-matching problem Γ admits a minimal complete set of E-matchers, which is $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$. It implies that not Γ is of type zero. By construction, the set $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ is infinite iff $\overline{x} \approx (\tilde{t})[f] \in \mathbf{LM}(\Gamma)$ or $\overline{x} \approx \{\!\{\tilde{t}\}\!\}[f] \in \mathbf{LM}(\Gamma)$ for some $\overline{x}$, $\tilde{t}$, and f. Inspecting the rules of $\mathbf{LM}$, one can see that such equations are in $\mathbf{LM}(\Gamma)$ iff Γ is an A- or an AC-matching problem. For syntactic and C-matching problems this is not the case. Hence, if Γ is a syntactic or a C-matching problem, $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ is finite. For some such Γ's this set may contain more than one element. For instance, $\mathbf{\Sigma}(\mathbf{LM}(\{f(\overline{x},\overline{y}) \ll_{\emptyset} f(a)\})) = \{\{\overline{x} \mapsto (), \overline{y} \mapsto a\}, \{\overline{x} \mapsto a, \overline{y} \mapsto ()\}\}$ and $\mathbf{\Sigma}(\mathbf{LM}(\{f(x,y) \ll_{\{\mathsf{C}(f)\}} f(a,b)\})) = \{\{x \mapsto a, y \mapsto b\}, \{x \mapsto b, y \mapsto a\}\}$. It implies that variadic linear syntactic and C-matching problems have at most finitary type.

On the other hand, when Γ is a linear A- or AC-matching problem, if it is solvable, then $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ can be either a singleton, a finite non-singleton, or an infinite set. As examples illustrating each of these possibilities we could take the following problems, where $E = \{\mathsf{A}(f)\}$ or $E = \{\mathsf{A}(f), \mathsf{C}(f)\}$:

- $\mathbf{\Sigma}(\mathbf{LM}(\{f(x) \ll_E f()\})) = \{\{x \mapsto f()\}\}$,
- $\mathbf{\Sigma}(\mathbf{LM}(\{f(x) \ll_E f(a)\})) = \{\{x \mapsto a\}, \{x \mapsto f(a)\}\}$,
- $\mathbf{\Sigma}(\mathbf{LM}(\{f(\overline{x}) \ll_E f(a)\})) = \{\{\overline{x} \mapsto (\tilde{r}_1, t, \tilde{r}_2)\} \mid \tilde{r}_1, \tilde{r}_2 \in \{f()\}^*, t \in \{a, f(a)\}\}$.

It implies that variadic linear A- or AC-matching problems have at most infinitary type. □

5. Nonlinear matching

To solve nonlinear variadic equational matching problems, we first linearize them, replacing multiple occurrences of the same variable by fresh variables. Then solve the obtained problem by the matching algorithm **LM** described in the previous section. The last step is to check the obtained solutions for consistency. We describe this step in this section.

We assume that the input is normalized with respect to associativity (the left hand sides) and associativity and commutativity (the right hand sides), whenever such symbols appear there.

Let S_{lin} be an answer computed by **LM** for the linearized version Γ_{lin} of a matching problem Γ. For each variable v in Γ we have variables $v_1, \ldots, v_n$ in Γ_{lin}, which correspond to different occurrences of v in Γ. We call them copied variables. The algorithm **RS** (for "reconstruct solutions") described below produces (nondeterministically) from S_{lin} a set S which solves Γ. We initialize S by S_{lin}.

RS: Reconstruct solutions.

Input: S_{lin}.

1. $S := S_{\mathsf{lin}}$.

2. Replace all copies of individual, function, and sequence variables in S with their original names. For instance, if S contains $v_x^1 \approx t_1$ and $v_x^2 \approx t_2$, replace them respectively by $x \approx t_1$ and $x \approx t_2$. If $t_1 = t_2$, this operation will collapse multiple occurrences of the same equation into one occurrence, because S is a set.

3. If S contains two equations for the same individual or function variable (i.e., two equations of the form $x \approx t_1$ and $x \approx t_2$ with $t_1 \neq t_2$, or $X \approx f_1$ and $X \approx f_2$ with $f_1 \neq f_2$), **stop** with failure.

4. If S contains two equations for the same sequence variable of the form $\overline{x} \approx \tilde{r}$, $\overline{x} \approx \tilde{t}$, or of the form $\overline{x} \approx \tilde{r}[f]$, $\overline{x} \approx \tilde{t}[f]$, where $\tilde{r} \neq \tilde{t}$, **stop** with failure.

5. If S contains two equations for the same sequence variable of the form $\overline{x} \approx \{\!\{\tilde{r}\}\!\}$, $\overline{x} \approx \{\!\{\tilde{t}\}\!\}$, or of the form $\overline{x} \approx \{\!\{\tilde{r}\}\!\}[f]$, $\overline{x} \approx \{\!\{\tilde{t}\}\!\}[f]$, where the multisets $\{\!\{\tilde{r}\}\!\}$ and $\{\!\{\tilde{t}\}\!\}$ are not equal, **stop** with failure.

6. Otherwise, if S contains two equations for the same sequence variable $\overline{x} \approx R_1$, $\overline{x} \approx R_2$, where

   (a) $R_1 = \tilde{r}$ or $R_1 = \tilde{r}[f]$, and $R_2 = \tilde{t}[g]$, or

   (b) $R_1 = \{\!\{\tilde{r}\}\!\}$ or $R_1 = \{\!\{\tilde{r}\}\!\}[f]$, and $R_2 = \tilde{t}$ or $R_2 = \tilde{t}[g]$, or

(c) $R_1 = \tilde{r}$ or $R_1 = \tilde{r}[f]$, and $R_2 = \{\!\{\tilde{t}\}\!\}$ or $R_2 = \{\!\{\tilde{t}\}\!\}[g]$, or

(d) $R_1 = \{\!\{\tilde{r}\}\!\}$ or $R_1 = \{\!\{\tilde{r}\}\!\}[f]$, and $R_2 = \{\!\{\tilde{t}\}\!\}[g]$,

with $f \neq g$, then create the pair $R_1 \sqcap R_2; R_0$, where $R_0 = ()$ in the cases (a)–(c) and $R_0 = \emptyset$ in the case (d), and apply the reduction rules in Group 1–4 below as long as possible. If the process ends with $\top; R$, then update S with $S := S \setminus \{\overline{x} \approx R_1, \overline{x} \approx R_2\} \cup \{\overline{x} \approx R\}$, and repeat Step 6. If the process terminates without reaching $\top; R$, **stop** with failure.

7. **Return** S.

The reduction rules operate on pairs of the form $R_1 \sqcap R_2; I$, where R_1 and R_2 originate from Step 6 of the **RS** algorithm above. (Remember that they are in normal form.) The goal is to compute a sequence (or a multiset) that "is common" between R_1 and R_2, i.e., some kind of "intersection" between them. More precisely, we aim at computing an I such that $\Sigma(I) = \Sigma(R_1) \cap \Sigma(R_2)$. We compute it in the second argument of the pair $R_1 \sqcap R_2; I$, which serves like an accumulator for the result computed so far during execution.

Due to the nature of R_1 and R_2, we have four groups of rules: when both R_1 and R_2 are sequences, when one of them is a sequence and the other one a multiset, and when both are multisets. In each group, there are four types of rules: success, common term/intersection, relax left, relax right. This gives 16 types of rules. Within each type, we have two, three, or four rules depending on whether the function symbols are attached to sequence/multisets or not.

The reduction rules are the following (in all rules $f \neq g$, and $\cup$, $\cap$, and $\setminus$ are operations on multisets):

Group 1, sequences in both sides. This group contains 10 rules:

(Succ-S) **Success, sequences:**

$$\begin{array}{lll} \text{Succ-S.1} & ()[f] \sqcap (); \ \tilde{r} & \Longrightarrow \top; \ \tilde{r} \\ \text{Succ-S.2} & () \sqcap ()[g]; \ \tilde{r} & \Longrightarrow \top; \ \tilde{r} \\ \text{Succ-S.3} & ()[f] \sqcap ()[g]; \ \tilde{r} & \Longrightarrow \top; \ \tilde{r} \end{array}$$

(CFT) **Common first term in sequences:**

CFT.1 $(r, \tilde{t}_1)[f] \sqcap (r, \tilde{t}_2);\ \tilde{r} \quad \Longrightarrow \tilde{t}_1[f] \sqcap \tilde{t}_2;\ (\tilde{r}, r)$

CFT.2 $(r, \tilde{t}_1) \sqcap (r, \tilde{t}_2)[g];\ \tilde{r} \quad \Longrightarrow \tilde{t}_1 \sqcap \tilde{t}_2[g];\ (\tilde{r}, r)$

CFT.3 $(r, \tilde{t}_1)[f] \sqcap (r, \tilde{t}_2)[g];\ \tilde{r} \Longrightarrow \tilde{t}_1[f] \sqcap \tilde{t}_2[g];\ (\tilde{r}, r)$

(RL-S) **Relax the left hand side, sequences:**

RL-S.1 $(g(\tilde{t}_1), \tilde{t}_2) \sqcap (\tilde{t}_1, \tilde{t}_3)[g];\ \tilde{r} \quad \Longrightarrow \tilde{t}_2 \sqcap \tilde{t}_3[g];\ (\tilde{r}, g(\tilde{t}_1))$

RL-S.2 $(g(\tilde{t}_1), \tilde{t}_2)[f] \sqcap (\tilde{t}_1, \tilde{t}_3)[g];\ \tilde{r} \Longrightarrow \tilde{t}_2[f] \sqcap \tilde{t}_3[g];\ (\tilde{r}, g(\tilde{t}_1))$

(RR-S) **Relax the right hand side, sequences:**

RR-S.1 $(\tilde{t}_1, \tilde{t}_2)[f] \sqcap (f(\tilde{t}_1), \tilde{t}_3);\ \tilde{r} \quad \Longrightarrow \tilde{t}_2[f] \sqcap \tilde{t}_3;\ (\tilde{r}, f(\tilde{t}_1))$

RR-S.2 $(\tilde{t}_1, \tilde{t}_2)[f] \sqcap (f(\tilde{t}_1), \tilde{t}_3)[g];\ \tilde{r} \Longrightarrow \tilde{t}_2[f] \sqcap \tilde{t}_3[g];\ (\tilde{r}, f(\tilde{t}_1))$

Example 7. We illustrate four rules: RL-S.2, RR-S.2, CFT.3, and Succ-S.3:

$(g(a, b), c)[f] \sqcap (a, b, f(), c)[g];\ () \Longrightarrow_{\text{RL-S.2}}$
$(c)[f] \sqcap (f(), c)[g];\ (g(a, b)) \Longrightarrow_{\text{RR-S.2}}$
$(c)[f] \sqcap (c)[g];\ (g(a, b), f()) \Longrightarrow_{\text{CFT.3}}$
$()[f] \sqcap ()[g];\ (g(a, b), f(), c) \Longrightarrow_{\text{Succ-S.3}}$
$\top;\ (g(a, b), f(), c).$

Note that both RL-S.2 and RR-S.2 can apply to some problem. Therefore, to get all solutions, we need to try both alternatives:

Alternative 1, leading to the result $(g(), f())$:

$(g())[f] \sqcap (f())[g];\ () \Longrightarrow_{\text{RL-S.2}}$
$()[f] \sqcap (f())[g];\ (g()) \Longrightarrow_{\text{RR-S.2}}$
$()[f] \sqcap ()[g];\ (g(), f()) \Longrightarrow_{\text{Succ-S.3}}$
$\top;\ (g(), f()).$

Alternative 2, leading to the result $(f(), g())$:

$(g())[f] \sqcap (f())[g];\ () \Longrightarrow_{\text{RR-S.2}}$
$(g())[f] \sqcap ()[g];\ (f()) \Longrightarrow_{\text{RL-S.2}}$
$()[f] \sqcap ()[g];\ (f(), g()) \Longrightarrow_{\text{Succ-S.3}}$
$\top;\ (f(), g()).$

Group 2, left multiset, right sequence. This group contains 12 rules:

(Succ-MS) **Success, multiset-sequence:**

$$\begin{array}{ll}
\textsf{Succ-MS.1} & \emptyset \sqcap (); \ \tilde{r} \qquad \Longrightarrow \top; \ \tilde{r} \\
\textsf{Succ-MS.2} & \emptyset[f] \sqcap (); \ \tilde{r} \quad \Longrightarrow \top; \ \tilde{r} \\
\textsf{Succ-MS.3} & \emptyset \sqcap ()[g]; \ \tilde{r} \quad \Longrightarrow \top; \ \tilde{r} \\
\textsf{Succ-MS.4} & \emptyset[f] \sqcap ()[g]; \ \tilde{r} \Longrightarrow \top; \ \tilde{r}
\end{array}$$

(CT-MS) **Common term, multiset-sequence:**

$$\begin{array}{ll}
\textsf{CT-MS.1} & (\{\!\{r\}\!\} \cup M) \sqcap (r, \tilde{t}); \ \tilde{r} \qquad \Longrightarrow M \sqcap \tilde{t}; \ (\tilde{r}, r) \\
\textsf{CT-MS.2} & (\{\!\{r\}\!\} \cup M) \ \sqcap (r, \tilde{t})[g]; \ \tilde{r} \quad \Longrightarrow M \sqcap \tilde{t}[g]; \ (\tilde{r}, r) \\
\textsf{CT-MS.3} & (\{\!\{r\}\!\} \cup M)[f] \sqcap (r, \tilde{t}); \ \tilde{r} \quad \Longrightarrow M[f] \sqcap \tilde{t}; \ (\tilde{r}, r) \\
\textsf{CT-MS.4} & (\{\!\{r\}\!\} \cup M)[f] \sqcap (r, \tilde{t})[g]; \ \tilde{r} \Longrightarrow M[f] \sqcap \tilde{t}[g]; \ (\tilde{r}, r)
\end{array}$$

(RL-MS) **Relax the left hand side, multiset-sequence:**

$$\begin{array}{ll}
\textsf{RL-MS.1} & (\{\!\{g(\tilde{t}_1)\}\!\} \cup M) \sqcap (\tilde{t}_1, \tilde{t}_2)[g]; \ \tilde{r} \\
& \qquad \Longrightarrow M \sqcap \tilde{t}_2[g]; \ (\tilde{r}, g(\tilde{t}_1)) \\
\textsf{RL-MS.2} & (\{\!\{g(\tilde{t}_1)\}\!\} \cup M)[f] \sqcap (\tilde{t}_1, \tilde{t}_2)[g]; \ \tilde{r} \\
& \qquad \Longrightarrow M[f] \sqcap \tilde{t}_2[g]; \ (\tilde{r}, g(\tilde{t}_1))
\end{array}$$

(RR-MS) **Relax the right hand side, multiset-sequence:**

$$\begin{array}{ll}
\textsf{RR-MS.1} & (\{\!\{\tilde{t}_1\}\!\} \cup M)[f] \sqcap (f(\tilde{t}_1), \tilde{t}_2); \ \tilde{r} \\
& \qquad \Longrightarrow M[f] \sqcap \tilde{t}_2; \ (\tilde{r}, f(\tilde{t}_1))) \\
\textsf{RR-MS.2} & (\{\!\{\tilde{t}_1\}\!\} \cup M)[f] \sqcap (f(\tilde{t}_1), \tilde{t}_2)[g]; \ \tilde{r} \\
& \qquad \Longrightarrow M[f] \sqcap \tilde{t}_2[g]; \ (\tilde{r}, f(\tilde{t}_1))
\end{array}$$

Example 8. We illustrate some of the rules in Group 2:

$$\begin{array}{l}
\{\!\{c, g(a,b), g()\}\!\} \sqcap (a, b, c, f())[g]; \ () \Longrightarrow_{\textsf{RL-MS.1}} \\
\{\!\{c, g()\}\!\} \sqcap (c, f())[g]; \ (g(a,b)) \Longrightarrow_{\textsf{CT-MS.2}} \\
\{\!\{g()\}\!\} \sqcap (f())[g]; \ (g(a,b), c) \Longrightarrow_{\textsf{RL-MS.1}}
\end{array}$$

$\{\!\{\}\!\} \sqcap (f())[g];\ (g(a,b), c, g())$.

Note that to the initial problem, we could also apply the RL-MS.1 differently, which would lead to a different derivation:

$\{\!\{c, g(a,b), g()\}\!\} \sqcap (a,b,c,f())[g];\ () \Longrightarrow_{\mathsf{RL\text{-}MS.1}}$
$\{\!\{c, g(a,b)\}\!\} \sqcap (a,b,c,f())[g];\ (g()) \Longrightarrow_{\mathsf{RL\text{-}MS.1}}$
$\{\!\{c\}\!\} \sqcap (c,f())[g];\ (g(), g(a,b)) \Longrightarrow_{\mathsf{CT\text{-}MS.2}}$
$\{\!\{\}\!\} \sqcap (f())[g];\ (g(), g(a,b), c)$.

Yet another alternative can be obtained from the first derivation by transforming $\{\!\{c, g()\}\!\} \sqcap (c,f())[g];\ (g(a,b))$ with RL-MS.1 instead of CT-MS.2, eventually leading to the terminal pair $\{\!\{\}\!\} \sqcap (f())[g];\ (g(a,b), g(), c)$.

Group 3, left sequence, right multiset. These rules are symmetric to those in the previous group, obtained by swapping the arguments of $\sqcap$. We do not list them explicitly. Their names are similar to the names of the rules in Group 2, with the difference that MS (multiset, sequence) is changed into SM (sequence, multiset).

Group 4, multisets in both sides. This group contains 10 rules:

(Succ-M) **Success, multisets:**

Succ-M.1 $\quad \emptyset[f] \sqcap \emptyset;\ M \quad \Longrightarrow \top;\ M$
Succ-M.2 $\quad \emptyset \sqcap \emptyset[g];\ M \quad \Longrightarrow \top;\ M$
Succ-M.3 $\quad \emptyset[f] \sqcap \emptyset[g];\ M \Longrightarrow \top;\ M$

(NMI) **Nonempty multiset intersection:**

NMI.1 $\quad M_1[f] \sqcap M_2;\ M$
$\qquad\qquad \Longrightarrow (M_1 \setminus N)[f] \sqcap (M_2 \setminus N);\ M \cup N$
NMI.2 $\quad M_1 \sqcap M_2[g];\ M$
$\qquad\qquad \Longrightarrow (M_1 \setminus N) \sqcap (M_2 \setminus N)[g];\ M \cup N$
NMI.3 $\quad M_1[f] \sqcap M_2[g];\ M$
$\qquad\qquad \Longrightarrow (M_1 \setminus N)[f] \sqcap (M_2 \setminus N)[g];\ M \cup N$

where $N = M_1 \cap M_2 \neq \emptyset$.

(RL-M) **Relax the left hand side, multisets:**

$$\begin{array}{ll}\mathsf{RL\text{-}M.1} & (\{\!\{g(\tilde{t}_1)\}\!\} \cup M_1) \sqcap (\{\!\{\tilde{t}_1\}\!\} \cup M_2)[g];\ M \\ & \quad \Longrightarrow M_1 \sqcap M_2[g];\ M \cup \{\!\{g(\tilde{t}_1)\}\!\} \\ \mathsf{RL\text{-}M.2} & (\{\!\{g(\tilde{t}_1)\}\!\} \cup M_1)[f] \sqcap (\{\!\{\tilde{t}_1\}\!\} \cup M_2)[g];\ M \\ & \quad \Longrightarrow M_1[f] \sqcap M_2[g];\ M \cup \{\!\{g(\tilde{t}_1)\}\!\}\end{array}$$

(RR-M) **Relax the left hand side, multisets:**

$$\begin{array}{ll}\mathsf{RR\text{-}M.1} & (\{\!\{\tilde{t}_1\}\!\} \cup M_1)[f] \sqcap (\{\!\{f(\tilde{t}_1)\}\!\} \cup M_2);\ M \\ & \quad \Longrightarrow M_1[f] \sqcap M_2;\ M \cup \{\!\{f(\tilde{t}_1)\}\!\} \\ \mathsf{RR\text{-}M.2} & (\{\!\{\tilde{t}_1\}\!\} \cup M_1)[f] \sqcap (\{\!\{f(\tilde{t}_1)\}\!\} \cup M_2)[g];\ M \\ & \quad \Longrightarrow M_1[f] \sqcap M_2[g];\ M \cup \{\!\{f(\tilde{t}_1)\}\!\}\end{array}$$

Example 9. We illustrate some of the rules in Group 4:

$$\begin{array}{l}\{\!\{a,b,c,a,b,c,d\}\!\}[f] \sqcap \{\!\{b,f(b,c),f(c,d),a,a\}\!\};\ \emptyset \Longrightarrow_{\mathsf{NMI.1}} \\ \{\!\{c,b,c,d\}\!\}[f] \sqcap \{\!\{f(b,c),f(c,d)\}\!\};\ \{\!\{a,a,b\}\!\} \Longrightarrow_{\mathsf{RR\text{-}M.1}} \\ \{\!\{c,d\}\!\}[f] \sqcap \{\!\{f(c,d)\}\!\};\ \{\!\{a,a,b,f(b,c)\}\!\} \Longrightarrow_{\mathsf{RR\text{-}M.1}} \\ \emptyset[f] \sqcap \emptyset;\ \{\!\{a,a,b,f(b,c),f(c,d)\}\!\} \Longrightarrow_{\mathsf{Succ\text{-}M.1}} \\ \top;\ \{\!\{a,a,b,f(b,c),f(c,d)\}\!\}.\end{array}$$

Note that alternative derivations in this case do not lead to different results, since the order of elements in the computed multiset does not matter. Therefore, it does not make sense to try rules in a different order. Nondeterminism between the rules in Group 4 is don't-care nondeterminism, unlike the previous three groups where it is don't-know nondeterminism.

Hence, as we saw, the rules RL-S.2 and RR-S.2 are alternatives of each other and can be used nondeterministically, when $\tilde{t}_1 = ()$. The same is true for RL-SM.2 and RR-SM.2, RL-MS.2 and RR-MS.2, and RL-M.2 and RR-M.2. There are also overlaps between common term rules and relax rules. All these are examples of don't-know nondeterminism. They cause branching in the derivation tree. On the other hand, nondeterminism in Group 4 is don't-care nondeterminism and we can fix an arbitrary application order of rules in that group. It does not cause branching.

The rules are applied as long as possible. The successful derivations are those that end with $\top; \tilde{r}$ for some sequence $\tilde{r}$ (as in Example 7), or with $\top; M$ for some multiset M (as in Example 9). Failed derivations are those which are not successful ones but no rule applies to them (as in Example 8). The set of all S's computed in Step 7 is denoted by $\mathbf{RS}(S_{\mathsf{lin}})$.

Example 10. Let $\{\mathsf{A}(f), \mathsf{C}(f), \mathsf{A}(g)\} = E$ and consider the matching problem

$$\Gamma = \{h(f(\overline{x}), g(\overline{x})) \ll_E h(f(a, g()), g(f(), a, f()))\}.$$

Linearization gives

$$\Gamma_{\mathsf{lin}} = \{h(f(\overline{x}_1), g(\overline{x}_2)) \ll_E h(f(a, g()), g(f(), a, f()))\},$$

where $\overline{x}_1$ and $\overline{x}_2$ are copies of $\overline{x}$. The algorithm **LM** returns one solved set

$$S_{\mathsf{lin}} = \{\overline{x}_1 \approx \{\!\{a, g()\}\!\}[f],\ \overline{x}_2 \approx (f(), a, f())[g]\}.$$

We apply the **RS** algorithm to S_{lin}. We restore the variable $\overline{x}$:

$$S := \{\overline{x} \approx \{\!\{a, g()\}\!\}[f],\ \overline{x} \approx (f(), a, f())[g]\}.$$

Step 6 of **RS** produces four alternative derivations:

$$\begin{aligned}
&\{\!\{a, g()\}\!\}[f] \sqcap (f(), a, f())[g]; () \Longrightarrow_{\mathsf{RL\text{-}MS.2}} \\
&\quad \{\!\{a\}\!\}[f] \sqcap (f(), a, f())[g]; g() \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \{\!\{a\}\!\}[f] \sqcap (a, f())[g]; (g(), f()) \Longrightarrow_{\mathsf{CT\text{-}MS.4}} \\
&\quad \emptyset[f] \sqcap (f())[g]; (g(), f(), a) \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \emptyset[f] \sqcap ()[g]; (g(), f(), a, f()) \Longrightarrow_{\mathsf{Succ\text{-}S.4}} \\
&\quad \top; (g(), f(), a, f()).
\end{aligned}$$

$$\begin{aligned}
&\{\!\{a, g()\}\!\}[f] \sqcap (f(), a, f())[g]; () \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \{\!\{a, g()\}\!\}[f] \sqcap (a, f())[g]; f() \Longrightarrow_{\mathsf{RL\text{-}MS.2}} \\
&\quad \{\!\{a\}\!\}[f] \sqcap (a, f())[g]; (f(), g()) \Longrightarrow_{\mathsf{CT\text{-}MS.4}} \\
&\quad \emptyset[f] \sqcap f()[g]; (f(), g(), a) \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \emptyset[f] \sqcap ()[g]; (f(), g(), a, f()) \Longrightarrow_{\mathsf{Succ\text{-}MS.4}}
\end{aligned}$$

$$\top; (f(), g(), a, f()).$$

$$\begin{aligned}
&\{\!\{a, g()\}\!\}[f] \sqcap (f(), a, f())[g]; () \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \{\!\{a, g()\}\!\}[f] \sqcap (a, f())[g]; f() \Longrightarrow_{\mathsf{CT\text{-}MS.4}} \\
&\quad \{\!\{g()\}\!\}[f] \sqcap (f())[g]; (f(), a) \Longrightarrow_{\mathsf{RL\text{-}MS.2}} \\
&\quad \emptyset[f] \sqcap f()[g]; (f(), a, g()) \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \emptyset[f] \sqcap ()[g]; (f(), a, g(), f()) \Longrightarrow_{\mathsf{Succ\text{-}MS.4}} \\
&\quad \top; (f(), a, g(), f()).
\end{aligned}$$

$$\begin{aligned}
&\{\!\{a, g()\}\!\}[f] \sqcap (f(), a, f())[g]; () \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \{\!\{a, g()\}\!\}[f] \sqcap (a, f())[g]; f() \Longrightarrow_{\mathsf{CT\text{-}MS.4}} \\
&\quad \{\!\{g()\}\!\}[f] \sqcap ()[g]; (f(), a) \Longrightarrow_{\mathsf{RR\text{-}MS.2}} \\
&\quad \{\!\{g()\}\!\}[f] \sqcap f()[g]; (f(), a, f()) \Longrightarrow_{\mathsf{RL\text{-}MS.2}} \\
&\quad \emptyset[f] \sqcap ()[g]; (f(), a, f(), g()) \Longrightarrow_{\mathsf{Succ\text{-}MS.4}} \\
&\quad \top; (f(), a, f(), g()).
\end{aligned}$$

These derivations give four solved forms as results of application of **RS** to S_{lin}. It is easy to see that they give solutions of Γ:

$$\begin{aligned}
\mathbf{RS}(S_{\mathsf{lin}}) := \{&\{\overline{x} \approx (g(), f(), a, f())\},\ \{\overline{x} \approx (f(), g(), a, f())\}, \\
&\{\overline{x} \approx (f(), a, g(), f())\},\ \{\overline{x} \approx (f(), a, f(), g())\}\}.
\end{aligned}$$

Example 11. Let $\{\mathsf{A}(f), \mathsf{A}(g)\} = E$ and consider the matching problem

$$\Gamma = \{h(f(\overline{x}, \overline{y}), f(\overline{y}), g(\overline{x})) \ll_E h(f(g(a), b, c, d), f(c, d), g(a, f(), f(b)))\}.$$

After linearizing it (taking $\overline{x}_1, \overline{x}_2$ as copies of $\overline{x}$ and $\overline{y}_1, \overline{y}_2$ as copies of $\overline{y}$) and using the algorithm **LM**, we get solved sets:

$$\begin{aligned}
S^1_{\mathsf{lin}} &= \{\overline{x}_1 \approx ()[f],\ \overline{y}_1 \approx (g(a), b, c, d)[f], \\
&\qquad \overline{y}_2 \approx (c, d)[f],\ \overline{x}_2 \approx (a, f(), f(b))[g]\}. \\
S^2_{\mathsf{lin}} &= \{\overline{x}_1 \approx g(a)[f],\ \overline{y}_1 \approx (b, c, d)[f],\ \overline{y}_2 \approx (c, d)[f], \\
&\qquad \overline{x}_2 \approx (a, f(), f(b))[g]\}. \\
S^3_{\mathsf{lin}} &= \{\overline{x}_1 \approx (g(a), b)[f],\ \overline{y}_1 \approx (c, d)[f],\ \overline{y}_2 \approx (c, d)[f], \\
&\qquad \overline{x}_2 \approx (a, f(), f(b))[g]\}. \\
S^4_{\mathsf{lin}} &= \{\overline{x}_1 \approx (g(a), b, c)[f],\ \overline{y}_1 \approx d[f],\ \overline{y}_2 \approx (c, d)[f],
\end{aligned}$$

$$\begin{aligned} & \overline{x}_2 \approx (a, f(), f(b))[g]\}. \\ S^5_{\mathsf{lin}} = \{&\overline{x}_1 \approx (g(a), b, c, d)[f],\ \overline{y}_1 \approx ()[f],\ \overline{y}_2 \approx (c, d)[f], \\ & \overline{x}_2 \approx (a, f(), f(b))[g]\}. \end{aligned}$$

Because of Step 4 of the algorithm, we get $\mathbf{RS}(S^1_{\mathsf{lin}}) = \mathbf{RS}(S^2_{\mathsf{lin}}) = \mathbf{RS}(S^4_{\mathsf{lin}}) = \mathbf{RS}(S^5_{\mathsf{lin}}) = \emptyset$. Step 6 gives $\mathbf{RS}(S^3_{\mathsf{lin}}) = \{\overline{x} \approx (g(a), f(), f(b)),\ \overline{y} \approx (c, d)[f]\}$, which is the final solution. The solved equation for $\overline{x}$ has been computed by the following reduction:

$$\begin{aligned} & (g(a), b)[f] \sqcap (a, f(), f(b))[g]; () \Longrightarrow_{\mathsf{RL\text{-}S.2}} \\ & \quad b[f] \sqcap (f(), f(b))[g]; g(a) \Longrightarrow_{\mathsf{RR\text{-}S.2}} \\ & \quad b[f] \sqcap f(b)[g]; (g(a), f()) \Longrightarrow_{\mathsf{RR\text{-}S.2}} \\ & \quad ()[f] \sqcap ()[g]; (g(a), f(), f(b)) \Longrightarrow_{\mathsf{Succ.3}} \\ & \quad \top; (g(a), f(), f(b)). \end{aligned}$$

Theorem 7 (Termination of **RS**)**.** *The procedure* **RS** *terminates for every input.*

Proof. Steps 2, 3, 4, and 5 in **RS** cannot cause non-termination. Step 2 is performed only once at the beginning, while 3, 4, and 5 immediately stop the procedure. As for Step 6, we cannot have an infinite sequence of applications of reduction rules, because each rule strictly reduces the size of the first component of the pair it transforms. Moreover, this step is performed either after Step 2, or after an application of itself. Hence, Step 6 is executed finitely many times, each time performing finitely many reduction steps. These observations imply termination of **RS**. □

Theorem 8 (Soundness and Completeness of **RS**)**.** *Let* $v \approx R_1$ *and* $v \approx R_2$ *be two equations for the same* v*, obtained from* S_{lin} *at Step 2 of the* **RS** *algorithm. Then* $\Sigma(v \approx R_1) \cap \Sigma(v \approx R_2) \neq \emptyset$ *iff either*

- $R_1 = R_2$*, or*
- $R_1 \neq R_2$ *and there exists a derivation* $R_1 \sqcap R_2; R_0 \Longrightarrow^* \top; R$ *in* **RS***, where* $R_0 = ()$ *or* $R_0 = \emptyset$*, and* $\Sigma(v \approx R_1) \cap \Sigma(v \approx R_2) = \Sigma(v \approx R)$*.*

Proof. If $v \in \mathcal{V}_{\mathrm{Fun}} \cup \mathcal{V}_{\mathrm{Ind}}$, then by the definition of Σ, we have $\Sigma(v \approx R_1) \cap \Sigma(v \approx R_2) \neq \emptyset$ iff $R_1 = R_2$. The **RS** algorithm detects it either in Step 2 or in Step 3. There is no attempt to construct a reduction derivation.

Now assume $v \in \mathcal{V}_{\mathrm{Seq}}$ and consider the possible forms of R_1 and R_2. They are in normal form, if we have symbols from the corresponding theories.

First, let both R_1 and R_2 be sequences. Recall that if they are of the form $\tilde{s}[f]$, then no term in $\tilde{s}$ has f as its head.

- $R_1 = \tilde{s}$, $R_2 = \tilde{t}$ for some $\tilde{s}$ and $\tilde{t}$, or $R_1 = \tilde{s}[f]$, $R_2 = \tilde{t}[f]$ for some $\tilde{s}$, $\tilde{t}$, and f. In this case, by the definition of Σ, we have $\Sigma(v \approx R_1) \cap \Sigma(v \approx R_2) \neq \emptyset$ iff $R_1 = R_2$. The **RS** algorithm detects it at Step 2 or at Step 4. No reduction derivation is constructed.

- $R_1 = \tilde{s}$, $R_2 = \tilde{t}[g]$ for some $\tilde{s}$, $\tilde{t}$, and associative g. Let Cond be the condition: $\tilde{s}$ is either $\tilde{t}$, or is obtained from $\tilde{t}$ by replacing some of its (possibly empty) subsequences $\tilde{t}'$ by $g(\tilde{t}')$. By the definition of Σ, we have $\Sigma(v \approx \tilde{s}) \cap \Sigma(v \approx \tilde{t}[g]) \neq \emptyset$ iff Cond holds. Moreover, when $\Sigma(v \approx \tilde{s}) \cap \Sigma(v \approx \tilde{t}[g]) \neq \emptyset$, then $\Sigma(v \approx \tilde{s}) \cap \Sigma(v \approx \tilde{t}[g]) = \{\{v \mapsto \tilde{s}\}\}$. The derivation constructed by **RS** for such R_1 and R_2 deals exactly with these cases using the rules Succ-S.2, CFT.2 and RL-S.1. It succeeds iff Cond holds, and fails otherwise. The R, obtained by a successful derivation, is $\tilde{s}$.

- $R_1 = \tilde{s}[f]$, $R_2 = \tilde{t}$ for some $\tilde{s}$, $\tilde{t}$, and associative f. The proof for this case is analogous to the previous one, using the rules the rules Succ-S.1, CFT-S.1 and RR-S.1 in the derivation.

- $R_1 = \tilde{s}[f]$, $R_2 = \tilde{t}[g]$ for some $\tilde{s}$, $\tilde{t}$, and associative f and g with $f \neq g$. Let Cond is this case be the condition consisting of four cases:

  1. $\tilde{s}$ and $\tilde{t}$ are the same, or
  2. $\tilde{s}$ is obtained from $\tilde{t}$ by replacing some of its (possibly empty) subsequences $\tilde{t}'$ by $g(\tilde{t}')$, or
  3. $\tilde{t}$ is obtained from $\tilde{s}$ by replacing some of its (possibly empty) subsequences $\tilde{s}'$ by $f(\tilde{s}')$, or
  4. in some places, where $\tilde{t}$ contains a (possibly empty) subsequence $\tilde{t}'$, $\tilde{s}$ contains $g(\tilde{t}')$, and vice versa: in some places, where $\tilde{s}$ contains a (possibly empty) subsequence $\tilde{s}'$, $\tilde{t}$ contains $f(\tilde{s}')$. ($\tilde{s}$ does not contain terms with the head f, and $\tilde{t}$ does not contain terms with the head g.)

By the definition of Σ, we have $\Sigma(v \approx \tilde{s}[f]) \cap \Sigma(v \approx \tilde{t}[g]) \neq \emptyset$ iff Cond holds. The derivation constructed by **RS** for such R_1 and R_2 deals exactly with these cases using the rules Succ-S.3, CFT-S.3, RL-S.2, and RR-S.2. It succeeds iff Cond holds, and fails otherwise. By applying RL-S.2 and RR-S.2 rules in different order we get different derivations. When successful, those derivations produce different elements of $\Sigma(v \approx \tilde{s}[f]) \cap \Sigma(v \approx \tilde{t}[g])$.

When at least one of R_1 and R_2 is a multiset, we will need to use the **RS** rules from groups 2, 3, or 4, in place of the rules from Group 1. The adaptation of the reasoning above to the corresponding cases is not difficult. □

Let $\mathbf{S}(\Gamma)$ be the set of all solved forms obtained for an E-matching problem Γ by **LM** and **RS**:

$$\mathbf{S}(\Gamma) := \{\mathbf{RS}(S_{\mathsf{lin}}) \mid S_{\mathsf{lin}} \in \mathbf{LM}(\Gamma_{\mathsf{lin}}), \Gamma_{\mathsf{lin}} \text{ is the linearization of } \Gamma\}.$$

Theorem 9. $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ *is a minimal complete set of E-matchers of an E-matching problem Γ.*

Proof. By Theorems 4 and 8, $\mathbf{\Sigma}(\mathbf{S})$ is a complete set of matchers for C-, A-, and AC- theories. For minimality, we reason as follows: First, our equational theories are regular, meaning that for each axiom $s \doteq t$ defining them we have $\mathcal{V}(s) = \mathcal{V}(t)$. Second, according to (Fages and Huet, 1986, Proposition 4.1), for a regular theory E, any complete set of different E-matchers of a term to a ground term is minimal. Redundancies in $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ can be caused only by substitutions that are equigeneral to each other (modulo C-, A- or AC, with respect to the variables of the input matching problem). This is because all input matching variables appear in the domain of each matcher in $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ (due to regularity), and the range of each matcher in $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ is ground (since matching problems have ground right hand sides). However, since our substitutions are in normal forms and the normal forms are unique, their equigenerality actually means their equality. Therefore, in $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ no two distinct substitutions are $\leq_E$-comparable. Hence, $\mathbf{\Sigma}(\mathbf{S}(\Gamma))$ is minimal. □

In Theorem 6 we showed what are the maximal matching types for linear C-, A-, and AC-problems. Now we get a more general result about matching types in these theories:

Theorem 10 (Matching types of variadic C-, A-, and AC-theories)**.**

- *The variadic* C*-theory has the finitary matching type.*
- *The variadic* A- *and* AC*-theories have the infinitary matching type.*

Proof. Follows from Theorem 6 and Theorem 9. □

Theorem 11. *Variadic associative, commutative, and associative-commutative matchings are NP-complete problems.*

Proof. First, we show that if an associative, commutative, or associative-commutative matching problem Γ is solvable, then it has a solution whose size is polynomially bounded by the size of Γ. For a linear Γ, this follows from the existence of a solution that just "fills the missing gaps" in the non-ground side to fit it to the ground one, without inserting extra terms of the form $f()$ in the instantiations of sequence variables. The exact construction of such a solution is given by algorithm **LM**. It shows that each occurrence of a non-associative symbol in the ground side of Γ (denoted by $\mathit{ground}(\Gamma)$) is consumed by at most one variable instantiation in the solution, while each occurrence of an associative (i.e., A or AC) symbol may appear in the instantiations of several variables. Hence, the size of such a matcher is bounded by $|\mathcal{V}(\Gamma)|\cdot|\mathit{ground}(\Gamma)|$, where $|\mathit{ground}(\Gamma)|$ is the size of the ground side of Γ.

For a nonlinear Γ, to show the existence of a polynomially bounded matcher, we can follow the construction of algorithm **RS**. The algorithm shows how to build solutions of Γ starting from the representation of solutions for its linearized version Γ_{lin}. For each function or individual variable in Γ, by this construction we have that any solution of Γ retains the representation of the instantiation of that variable in the solution of Γ_{lin}. For a sequence variable $\overline{x} \in \mathcal{V}(\Gamma)$, the size of its instantiation in a solution of Γ can be, in the worst case, the sum of the instantiation sizes for its copies in Γ_{lin}. The worst case is achieved for A- or AC-matching problems, e.g., of the form $\Gamma = \{f(g(\overline{x}), h(\overline{x})) \ll_E f(g(h()^n), h(g()^m))\}$, where $\{\mathsf{A}(g), \mathsf{A}(h)\} \subseteq E$ and $h()^n$ stands for the n-element sequence $(h(), \ldots, h())$. (Same for $g()^m$.) Then for Γ_{lin}, by **LM** we get a solution $\{v^1_{\overline{x}} \approx (h()^n)[g], v^2_{\overline{x}} \approx (g()^m)[h]\}$, from which **RS** gives $n+m$-long sequences, e.g. $\overline{x} \approx (h()^n, g()^m)$.

Hence, we showed that a solvable Γ has a matcher whose size is bounded polynomially in the size of Γ, by $\sum_{v\in\mathcal{V}(\Gamma)} \#(v,\Gamma)\cdot|\mathit{ground}(\Gamma)|$, where $\#(v,\Gamma)$ is the number of occurrences of v in Γ.

Now membership in NP follows from the fact that checking whether a substitution is a solution of any of stated matching problems can be done in polynomial time. The check involves the normalization of a term with respect to associativity and commutativity (according to the given ordering on function symbols) and then checking two terms for equality.

Hardness follows from the hardness of the syntactic variadic matching problem. The latter can be shown by a reduction from positive 1-in-3-SAT (Schaefer, 1978). In positive 1-in-3-SAT problems, clauses contain three positive literals. One looks for a truth assignment which makes exactly one literal true in each clause. In the reduction, we associate a term variable to each literal and encode a clause $p_1 \vee p_2 \vee p_3$ as a matching problem

$$assign(\overline{y}_1, or(x_1, x_2, x_3), \overline{y}_2) \ll assign(or(t, f, f), or(f, t, f), or(f, f, t)),$$

where the variable x_i corresponds to p_i, $1 \leq i \leq 3$, the symbol t corresponds to true, and the symbol f to false. Each solution to such a matching problem corresponds to an 1-in-3 truth assignment. For instance, $\{\overline{y}_1 \mapsto (), x_1 \mapsto t, x_2 \mapsto f, x_3 \mapsto f, \overline{y}_2 \mapsto (or(f, t, f), or(f, f, t))\}$ corresponds to the truth assignment that maps p_1 to true and p_2 and p_3 to false. This encoding is similar to the one used in proving NP-hardness of regular expression order-sorting matching (Kutsia and Marin, 2015).

To encode a SAT problem, i.e., a set of clauses, we take matching equations $s_1 \ll_E t_1, \ldots, s_k \ll_E t_k$ for the clauses and form a single matching equation as usual, using a free function symbol: $g(s_1, \ldots, s_k) \ll_E g(t_1, \ldots, t_k)$. The encoding is polynomial and preserves solvability in both directions: for each matcher, there exists the corresponding truth assignment that solves the given positive 1-in-3 SAT problem and vice versa. It implies that syntactic variadic matching problem is NP-hard. □

We conclude this section with a remark on a potential application of our finite representation of matchers' sets in rule-based transformations modulo variadic equational theories. To make such transformations work, one could enrich rules with constraints, which represent solved set equations. At each step, matching will need to take into account the existing constraints. It goes beyond the scope of this paper and can be a subject of a potential future work.

6. Finitary fragment and variant

A *fragment* of equational matching is obtained by restricting the form of the input, while *variants* require computing solutions of some special form without restricting the input. The algorithms **LM** and **RS** provide a finite representation of potentially infinite complete sets of E-matchers. An interesting question is to identify special cases when the set itself is finite. In this section we discuss two such special cases: the bounded fragment and the strict variant.

6.1. Bounded fragment

We start with a fragment that restricts occurrences of sequence variables.

Definition 5. Let Γ be a matching problem. A sequence variable $\overline{x}$ is called *bounded* in Γ if

- it occurs as an argument of at least two different function symbols or
- it is not an argument of any associative and associative-commutative symbol in Γ.

The problem Γ is called *bounded* if all its sequence variables are bounded in it.

Example 12. Let $E = \{\mathsf{A}(f), \mathsf{C}(f), \mathsf{A}(g), \mathsf{C}(g)\}$. The following matching problems are bounded:

- $\{f(\overline{x}) \ll_E f(a, g(b)),\ g(\overline{x}) \ll_E g(f(a), b)\}$, which has two solutions $\{\overline{x} \mapsto (f(a), g(b))\}$ and $\{\overline{x} \mapsto (g(b), f(a))\}$.
- $\{f(\overline{x}) \ll_E f(g(), g()),\ g(\overline{x}) \ll_E g(f(), f(), f())\}$, which has 10 solutions: $\{\{\overline{x} \mapsto \tilde{t}\} \mid \tilde{t}$ is a permutation of $(g(), g(), f(), f(), f())\}$.

An important property of bounded matching problems is the existence of a bound on the size of their solutions. More precisely, the following lemma holds:

Lemma 2. *Let Γ be a bounded matching problem and σ be its solution. Assume that the terms in the range of σ are A-normalized. Then for every variable $v \in \mathcal{V}(\Gamma)$, we have $size(v\sigma) \leq \sum_{s \ll_E t \in \Gamma} size(t)$.*

Proof. Since Γ is bounded, there will be an occurrence of v in some $s \ll_E t \in \Gamma$ such that $v\sigma$ is not flattened in $s\sigma$. If the inequality does not hold, we will have $size(s\sigma) \geq size(v\sigma) > size(t)$, contradicting the assumption that σ solves Γ. □

For bounded problems, **RS** returns a solution in which solved equations for sequence variables have the form $\overline{x} \approx \tilde{r}$, but neither $\overline{x} \approx \tilde{r}[f]$ nor $\overline{x} \approx M[f]$. This is easy to see: if a sequence variable occurs only under free or commutative symbol, then **LM** never applies the rules SVE-A and SVE-AC, which is the only source of generating solved equations of the form $\overline{x} \approx \tilde{r}[f]$ or $\overline{x} \approx M[f]$. If a sequence variable, say $\overline{x}$, appears under two distinct function symbols, say f and g, it can be that those symbols are, e.g., associative and after **LM** we have two equations of the form $\overline{y} \approx \tilde{s}[f]$ and $\overline{z} \approx \tilde{t}[g]$, where $\overline{y}$ and $\overline{z}$ are copies of $\overline{x}$. However, application of **RS** to $\tilde{s}[f] \sqcap \tilde{t}[g]$ produces $\tilde{r}$ without a function symbol attached to it, which gives a solved equation $\overline{x} \approx \tilde{r}$. (The same reasoning applies to the associative-commutative case.)

Hence, for bounded problems, no solution computed by **LM** and **RS** contains an equation of the form $\overline{x} \approx \tilde{r}[f]$ or $\overline{x} \approx M[f]$. It implies that if S is a solution of a bounded problem Γ, the set $\Sigma(S)$ is finite. Moreover, the number of distinct solutions is always finite for any matching problem. Therefore, the set of all solutions $\mathbf{S}$ of Γ is finite, and $\mathbf{\Sigma}(\mathbf{S})$ is finite.

6.2. Strict variant

Infinitely many solutions to our matching problems are caused by sequences of $f()$'s in the matchers, where f is an A or AC function symbol. But one might be interested in solutions, which do not introduce such extra $f()$'s.

For a precise characterization, we modify the variadic associativity axiom into variadic strict associativity:

$$f(\overline{x}, f(\overline{y}_1, y, \overline{y}_2), \overline{z}) \doteq f(\overline{x}, \overline{y}_1, y, \overline{y}_2, \overline{z}).$$

For an f, this axiom is denoted by $\mathsf{A}_\mathsf{s}(f)$ and we use A_s for the corresponding equational theory. Obviously, any solution of a matching problem modulo A_s or $\mathsf{A}_\mathsf{s}\mathsf{C}$ is also a solution modulo A or AC. Hence, we can say that we are aiming at solving a variant of A or AC-matching. We call it the *strict* variant and adapt our algorithms to compute matchers for it. Actually, the adaptation is pretty small. It concerns the definition of Σ and an extra condition for the A and AC rules in **LM** and for relaxing rules in **RS**. They

are discussed below. The notions of $\mathsf{A_s}$-normal form and $\mathsf{A_sC}$-normal form are defined analogously to their A- and AC-counterparts.

Adapting the definition of Σ *to* $\mathsf{A_s}$. In solved sets, the equations of the form $\overline{x} \approx (t_1, \ldots, t_n)[f]$ and $\overline{x} \approx \{\!\!\{t_1, \ldots, t_n\}\!\!\}[f]$ now will have f as a strict associative symbol (instead of associative). Consequently, in the definition of Σ, $\approx_{\mathsf{A}}$ is changed into $\approx_{\mathsf{A_s}}$ and A- and AC-normal forms into $\mathsf{A_s}$- and $\mathsf{A_sC}$-normal forms, respectively.

This modified definition defines a *finite* set of pairwise $\leq_E^{\{\overline{x}\}}$-incomparable substitutions. For instance, $\Sigma(\overline{x} \approx (t_1, t_2, t_3)[f])$ for a strict associative f is the set

$$\begin{aligned}
&\{\{\overline{x} \mapsto (r_1, r_2, r_3)\} \mid r_j \in \{t_j, f(t_j)\},\ 1 \leq j \leq 3\} \\
\cup\, &\{\{\overline{x} \mapsto (f(t_1, t_2), r_3)\} \mid r_3 \in \{t_3, f(t_3)\}\} \\
\cup\, &\{\{\overline{x} \mapsto (r_1, f(t_2, t_3))\} \mid r_1 \in \{t_1, f(t_1)\}\} \\
\cup\, &\{\{\overline{x} \mapsto (f(t_1, t_2, t_3))\}\}.
\end{aligned}$$

Adapting the associative and associative-commutative rules to $\mathsf{A_s}$. We replace the associative and associative-commutative rules by their strict counterparts, requiring strict associativity in the conditions. In addition, in the FVE-A-strict and FVE-AC-strict rules we require $\tilde{s} \neq ()$, in the IVE-A-strict rule $\tilde{t}_1 \neq ()$, and in the IVE-AC-strict rule $n > 0$. For SVE-A-strict and SVE-AC-strict, we do not need any extra condition, because the modified definition of Σ does the job. We call the modified algorithm $\mathbf{LM_S}$.

Adapting the reduction rules to $\mathsf{A_s}$. This change is motivated by the fact that in the relaxing rules, one cannot relax the sides by pretending that terms like $f()$ or $g()$ stand in the corresponding sequences or multisets. We need to impose an extra condition on those rules. Namely, we require $\tilde{t}_1 \neq ()$ in each of the relaxing rules. We call the modified algorithm $\mathbf{RS_S}$.

It is assumed that the equations these algorithms work on are normalized: $\mathbf{LM_S}$ works on equations with $\mathsf{A_s}$-normal left hand sides and $\mathsf{A_sC}$-normal right hand sides, and $\mathbf{RS_S}$ on $\mathsf{A_sC}$-normal forms.

Theorem 12. *The algorithm* $\mathbf{LM_S}$ *is terminating, sound, complete, and minimal.*

Proof. Termination is obvious, since **LM** is terminating (Theorem 2). Soundness means that the algorithm only computes strict solutions. That it computes solutions, follows from Theorem 3. Strictness of the computed solutions follows from the fact that no rule invents terms like $f()$ for the matchers. Completeness follows from Theorem 4 based on the observation that the new definition of Σ and the new conditions in the rules do not discard strict solutions. Hence, $\mathbf{LM}_{\mathrm{S}}$ computes all linear strict solutions and, therefore, is complete. Minimality means the minimality of the set $\mathbf{\Sigma}(\mathbf{LM}_{\mathrm{S}}(\Gamma))$ for any Γ, which follows from the minimality of $\mathbf{\Sigma}(\mathbf{LM}(\Gamma))$ (Theorem 5). □

Similarly, **RS** remains terminating, sound, and complete for theories involving A_{s}:

Theorem 13. *The algorithm* $\mathbf{RS}_{\mathrm{S}}$ *is terminating, sound, and complete.*

Proof. $\mathbf{RS}_{\mathrm{S}}$ terminates, since **RS** terminates (Theorem 7). Soundness and completeness can be shown similarly to soundness and completeness of **RS**. We need to take into account the definition of Σ adapted to A_{s} and the new conditions imposed over the relaxing reduction rules, which do not allow to introduce extra subsequences of the form $(f(), \ldots, f())$ in solutions, preventing non-strict results. □

Example 13. If $E = \{\mathsf{A}(f)\}$, the matching problem $f(\overline{x}) \ll_E f(a, a)$ has infinitely many solutions. If $E = \{\mathsf{A}_{\mathrm{s}}(f)\}$, there are five matchers: $\{\overline{x} \mapsto (a, a)\}$, $\{\overline{x} \mapsto f(a, a)\}$, $\{\overline{x} \mapsto (f(a), a)\}$, $\{\overline{x} \mapsto (a, f(a))$, $\{\overline{x} \mapsto (f(a), f(a))\}$.

Note that the size of matchers is bounded by the size of the right-hand side of matching equations both for the bounded fragment and for the strict variant.

7. Experimenting with the Mathematica variant

7.1. Data and observations

The programming language of Mathematica, called Wolfram, supports equational variadic matching in A, C, AC theories with individual and sequence variables. The terminology is a bit different, though. Variadic associative symbols there are called *flat* and commutative ones *orderless*. Individual variables correspond to patterns like `x_`, and sequence variables to patterns like `y___`.

The matching variants used in Mathematica are efficiently implemented, but the algorithm is not public. In this section we first show Mathematica's behavior on some selected characteristic examples and then will try to imitate it by variants of our rules. In the experiments we used the Mathematica built-in function `ReplaceList[expr,rules]`, which attempts to transform the entire expression `expr` by applying a rule or list of rules in all possible ways, and returns a list of the results obtained. In transformation, Mathematica tries to match `rules` to `expr`, exhibiting the behavior of the built-in matching mechanism. The equational theories can be specified by setting attributes (flat, orderless) to symbols.

The examples below are used to illustrate the behavior of Mathematica, but we prefer to write those examples in the notation of this paper. We compare it to our strict variant, because it also does not compute extra $f()$'s in the answer. However, they are not the same, as the examples below show. We report only sets of matchers, ignoring their order and how many times the same (syntactically or modulo an equational theory) matcher was computed. The strict variant treats associativity of input symbols as strict associativity. Mathematica treats them as symbols with the attribute `Flat`.

Problem:	$f(\overline{x}) \ll_E f(a),\ f$ is A or AC.	(5)
Strict matchers:	$\{\overline{x} \mapsto a\}, \{\overline{x} \mapsto f(a)\}$.	
Mathematica:	$\{\overline{x} \mapsto a\}$.	
Problem:	$f(f()) \ll_E f(),\ f$ is A or AC.	(6)
Strict matchers:	No solutions.	
Mathematica:	ε.	
Problem:	$f(x, y) \ll_E f(a, b),\ f$ is AC.	(7)
Strict matchers:	$\{x \mapsto t_1, y \mapsto t_2\}$, with $t_1 \in \{a, f(a)\}, t_2 \in \{b, f(b)\}$, $\{x \mapsto t_1, y \mapsto t_2\}$, with $t_1 \in \{b, f(b)\}, t_2 \in \{a, f(a)\}$.	
Mathematica:	$\{x \mapsto f(a), y \mapsto f(b)\}, \{x \mapsto a, y \mapsto b\}$, $\{x \mapsto f(b), y \mapsto f(a)\}, \{x \mapsto b, y \mapsto a\}$	
Problem:	$f(\overline{x}, \overline{y}) \ll_E f(a, b),\ f$ is AC.	(8)
Strict matchers:	$\{\overline{x} \mapsto (), \overline{y} \mapsto (t_1, t_2)\}$,	

	with $t_1 \in \{a, f(a)\}, t_2 \in \{b, f(b)\}$,	
	$\{\overline{x} \mapsto (), \overline{y} \mapsto (t_1, t_2)\}$,	
	with $t_1 \in \{b, f(b)\}, t_2 \in \{a, f(a)\}$,	
	$\{\overline{x} \mapsto (), \overline{y} \mapsto f(a, b)\}$,	
	$\{\overline{x} \mapsto (t_1, t_2), \overline{y} \mapsto ()\}$,	
	with $t_1 \in \{a, f(a)\}, t_2 \in \{b, f(b)\}$,	
	$\{\overline{x} \mapsto (t_1, t_2), \overline{y} \mapsto ()\}$,	
	with $t_1 \in \{b, f(b)\}, t_2 \in \{a, f(a)\}$,	
	$\{\overline{x} \mapsto f(a, b), \overline{y} \mapsto ()\}$,	
	$\{\overline{x} \mapsto t_1, \overline{y} \mapsto t_2\}$,	
	with $t_1 \in \{a, f(a)\}, t_2 \in \{b, f(b)\}$,	
	$\{\overline{x} \mapsto t_1, \overline{y} \mapsto t_2\}$,	
	with $t_1 \in \{b, f(b)\}, t_2 \in \{a, f(a)\}$,	
Mathematica:	$\{\overline{x} \mapsto a, \overline{y} \mapsto b\}, \{\overline{x} \mapsto b, \overline{y} \mapsto a\}$,	
	$\{\overline{x} \mapsto (a, b), \overline{y} \mapsto ()\}, \{\overline{x} \mapsto (), \overline{y} \mapsto (a, b)\}$,	
Problem:	$f(x, \overline{y}) \ll_E f(a, b, c)$, f is A.	(9)
Strict matchers:	$\{x \mapsto a, \overline{y} \mapsto f(b, c)\}, \{x \mapsto f(a), \overline{y} \mapsto f(b, c)\}$,	
	$\{x \mapsto f(a, b), \overline{y} \mapsto c\}, \{x \mapsto f(a, b), \overline{y} \mapsto f(c)\}$.	
	$\{x \mapsto t_1, \overline{y} \mapsto (t_2, t_3)\}$, with $t_1 \in \{a, f(a)\}$,	
	$t_2 \in \{b, f(b)\}, t_3 \in \{c, f(c)\}$,	
	$\{x \mapsto f(a, b, c), \overline{y} \mapsto ()\}$.	
Mathematica:	$\{x \mapsto a, \overline{y} \mapsto (b, c)\}, \{x \mapsto f(a), \overline{y} \mapsto (b, c)\}$,	
	$\{x \mapsto f(a, b), \overline{y} \mapsto c\}, \{x \mapsto f(a, b, c), \overline{y} \mapsto ()\}$.	
Problem:	$g(f(\overline{x}), \overline{x}) \ll_E g(f(a), f(a))$,	
	f is A or AC, g is free.	(10)
Strict matchers:	$\{\overline{x} \mapsto f(a)\}$.	
Mathematica:	No solutions.	
Problem:	$g(f(\overline{x}), g(\overline{x})) \ll_E g(f(b, a), g(b, a))$,	
	f is C, g is free.	(11)
Strict matchers:	$\{\overline{x} \mapsto (b, a)\}$.	
Mathematica:	No solutions.	

In (6), strictness does not allow one to flatten the left hand side, but Mathematica does not have this restriction and transforms the term into $f()$.

Interestingly, the behavior of Mathematica's matching changed from Version 6.0 to Version 11.2.[3] As reported in (Kutsia, 2008), for Problem (9), Mathematica 6.0 would return three out of four substitutions reported above. It would not compute $\{x \mapsto a, \overline{y} \mapsto (b, c)\}$.

For problems like (7), Mathematica does not compute a matcher in which one individual variable is mapped to a subterm from the right hand side, and the other one is instantiated by f applied to a *single* subterm from the right hand side. (The same is true when f is A.) Examining more examples, e.g. $f(x, y, z) \ll_E f(a, b, c, d)$, for f being AC, confirms this observation. One can see there matchers like $\{x \mapsto a, y \mapsto b, z \mapsto f(c, d)\}$ and $\{x \mapsto f(a), y \mapsto f(b), z \mapsto f(c, d)\}$ (and many more, the solution set consists of 72 matchers), but not $\{x \mapsto a, y \mapsto f(b), z \mapsto f(c, d)\}$ or similar.

However, this concerns only those individual variables which are arguments of the same occurrence of an associative symbol, as in $f(x, y) \ll_E f(a, b)$. For problems like $g(f(x), f(y)) \ll_E g(f(a), f(b))$, where g is free and f is associative or associative-commutative, Mathematica computes mixed solutions: $\{x \mapsto f(a), y \mapsto f(b)\}$, $\{x \mapsto f(a), y \mapsto b\}$, $\{x \mapsto a, y \mapsto f(b)\}$, $\{x \mapsto a, y \mapsto b\}$.

It is interesting to see how sequence variables behave in such a situation. Example (8) shows that in Mathematica matchers, f is not applied to terms from the right hand side. Besides, when a sequence variable is instantiated by a sequence, the order of elements in that sequence coincide with their relative order in the *canonical form* of the right hand side of the equation. For instance, in (8) Mathematica does not return $\{\overline{x} \mapsto (b, a), \overline{y} \mapsto ()\}$. Note that if f were only C in (8), Mathematica would still compute exactly the same set of matchers.

The canonical form of the right hand side is important. There is a so

[3]The experiments with pattern matching described in this paper have been originally carried out for Mathematica 11.2. Later, we repeated them for Mathematica 12.0 and got the same results.

called canonical order imposed on all Mathematica expressions,[4] and whenever there is a commutative symbol, the system rearranges its arguments according to this order. This is why Mathematica returns $\{\overline{x} \mapsto (a, b)\}$ to the matching problem $f(\overline{x}) \ll_E f(b, a)$, when $\mathsf{C}(f) \in E$. The arguments of $f(b, a)$ are first rearranged by the canonical order into $f(a, b)$, and matching is performed afterwards. These issues affect solvability of problems, as one can see in (11). It also indicates that imitating Mathematica's nonlinear matching (i.e. when the same variable occurs more than once in matching equations) is not trivial.

The final set of examples concerns function variables. One can see from the examples that even if a function variable gets instantiated by an equational symbol, Mathematica treats that instance as a free symbol (in (14) and (15), $f(b, a)$ is first normalized to $f(a, b)$):

Problem:	$X(x) \ll_E f(a)$, f is A or AC.	(12)
Strict matchers:	$\{X \mapsto f, x \mapsto a\}, \{X \mapsto f, x \mapsto f(a)\}$.	
Mathematica:	$\{X \mapsto f, x \mapsto a\}$.	
Problem:	$X(x) \ll_E f(a, b)$, f is A or AC.	(13)
Strict matchers:	$\{X \mapsto f,\ x \mapsto f(a, b)\}$.	
Mathematica:	No solutions.	
Problem:	$X(a, b) \ll_E f(b, a)$, f is C.	(14)
Strict matchers:	$\{X \mapsto f\}$.	
Mathematica:	$\{X \mapsto f\}$.	
Problem:	$X(b, a) \ll_E f(b, a)$, f is C.	(15)
Strict matchers:	$\{X \mapsto f\}$.	
Mathematica:	No solutions.	
Problem:	$f(x, X(b, c)) \ll_E f(a, b, c), f$ is A or AC.	(16)
Strict matchers:	$\{X \mapsto f\}$.	
Mathematica:	No solutions.	

[4] Roughly, the canonical order orders symbols alphabetically and extends to trees with respect to left-to-right pre-order.

7.2. Imitating variadic equational matching of Mathematica

These observations suggest that Mathematica's equational matching can be characterized as an incomplete strict variant of a normalized fragment. It means that in any combination of A and C (flat and orderless) theories, the matching algorithm takes input that is in the normal form (with respect to the given theory, without the strictness assumption) and aims at computing strict matchers for it. The algorithm is not complete. We suppose that it is a deliberate decision made for efficiency. It can happen that the same matcher is computed several times, supposedly in different ways of applying the algorithm.

One can also observe that an algorithm that tries to imitate Mathematica's equational matching behavior should indeed follow the idea of our construction: first linearizing the matching problem, then solving the obtained linear problem, and finally reconstructing the solution of the original one. For the exact details, we need to look into the behavior of each kind of variable. This is done below. Note that the original given matching problem is assumed to be A-/C-/AC-normalized.

Imitating the behavior of function variables. Equations of the form $X(\tilde{s}) \ll_E f(\tilde{t})$ are transformed by a new rule, which we denote by FVE-M (M for modified): $X(\tilde{s}) \ll_E f(\tilde{t}) \rightsquigarrow_\vartheta \{g(\tilde{s}) \ll_E g(\tilde{t})\}$, where $\vartheta = \{X \mapsto f\}$ and g is free. This rule replaces FVE. Besides, FVE-A-strict and FVE-AC-strict are dropped.

Imitating the behavior of individual and sequence variables under commutative symbols. The rule SVE-C is dropped. Instead, SVE-F is allowed to be used. (As we will see below, SVE-F can be used actually without any restriction to the head of the involved terms and, therefore, we can rename it to SVE.) On the other hand, Dec-C stays. Hence, commutative terms with individual variables can still be decomposed by the Dec-C rule.

Imitating the behavior of sequence variables under A_s *and* $\mathsf{A}_\mathrm{s}\mathsf{C}$ *symbols.* For equations $f(\tilde{s}) \ll_E f(\tilde{t})$, where $\mathsf{A}_\mathrm{s}(f) \in E$ and a sequence variable $\overline{x}$ appears in its solution σ, no element of the sequence $\overline{x}\sigma$ should have f as its head. For this, to make things simple, we just drop SVE-A-strict and SVE-AC-strict. Instead, SVE-F may apply also to strict associative and strict associative-commutative heads. We said above that this rule applies also to terms with a commutative head. We rename this modified rule into SVE, to reflect that it is not restricted to free symbols only and applies to any head. It implies that for the algorithm that reconstructs nonlinear solutions from the linear

ones, we will not have the reduction rules anymore (i.e., Step 6 of $\mathbf{RS}_{\mathrm{S}}$ does not apply).

Imitating the behavior of individual variables under A_{s} *and* $\mathsf{A}_{\mathrm{s}}\mathsf{C}$ *symbols.* This concerns equations of the form $f(\tilde{s}) \ll_E f(\tilde{t})$, where $\mathsf{A}_{\mathrm{s}}(f) \in E$. As we observed, if individual variables x and y occur as arguments of $f(\tilde{s})$, and t_1 and t_2 are two terms among $\tilde{t}$, then for the same matcher σ it cannot happen that $x\sigma = t_1$ and $y\sigma = f(t_2)$: Either $x\sigma$ should also have the head f, or $y\sigma$ should have more arguments. This is what Mathematica does.

To imitate this behavior, we introduce markings for equations of the form $s \ll_E t$, where $\mathit{head}(s) = \mathit{head}(t)$ and $\mathsf{A}_{\mathrm{s}}(\mathit{head}(s)) \in E$. Initially, they are not marked. First, assume that f is not commutative.

If for an unmarked equation $s \ll_E t$, the first argument of s is not an individual variable, then no marking takes place.

Now assume that an unmarked equation has a form $f(x, \tilde{s}) \ll_E f(t, \tilde{t})$, where f is A_{s}. There are only two rules that apply to this equation: Dec-A-strict and IVE-A-strict. Then marking works in the following way:

- If the unmarked equation is transformed by the Dec-A-strict rule into $\{x \ll_E t, f(\tilde{s}) \ll_E f(\tilde{t})\}$, then $f(\tilde{s}) \ll_E f(\tilde{t})$ is marked by 0.
- If the unmarked equation is transformed by the IVE-A-strict rule into $\{x \ll_E f(t), f(\tilde{s}) \ll_E f(\tilde{t})\}$, then $f(\tilde{s}) \ll_E f(\tilde{t})$ is marked by 1.
- Otherwise, if the unmarked equation is transformed by the IVE-A-strict rule into $\{x \ll_E f(t, \tilde{t}_1), f(\tilde{s}) \ll_E f(\tilde{t}_2)\}$, where $\tilde{t}_1 \neq ()$, then $f(\tilde{s}) \ll_E f(\tilde{t}_2)$ remains unmarked.

After introducing markings, they are used in rule applications. Remember that for each marked equation $s \ll_E t$ we have $\mathit{head}(s) = \mathit{head}(t)$ and $\{\mathsf{A}_{\mathrm{s}}(\mathit{head}(s))\} = E$. Therefore, the applicable rules are T, SVE, Dec-A-strict, or IVE-A-strict.

- If T applies, the equation is removed.
- If SVE applies (i.e., if the first argument of s is a sequence variable), then the obtained equation retains the marking of $s \ll_E t$.
- Otherwise, if the first argument of s is not an individual variable, the only applicable rule is Dec-A-strict. It produced two new equations. The first one, obtained from the first arguments of s and t, is unmarked. The second one retains the marking of $s \ll_E t$.

- Otherwise, the first argument of s is an individual variable. We have two cases depending on the marking of $s \ll_E t$:
  - $s \ll_E t$ is marked by 1. Then it will not be transformed by the Dec-A-strict rule. Only IVE-A-strict applies and the obtained equation retains the mark 1.
  - $s \ll_E t$ is marked by 0. Then it can be transformed both by Dec-A-strict and IVE-A-strict. If Dec-A-strict is used, two equations are obtained, from which the second one retains the mark 0. If IVE-A-strict is used, the sequence $\tilde{t}_1$ in the rule cannot be a singleton: more terms should be put in it. Also here, the obtained equation has the mark 0.

Marking works analogously for $\mathsf{A_sC}$ matching: simply use the Dec-AC-strict and IVE-AC-strict rules instead of Dec-A-strict and IVE-A-strict.

The algorithm $\mathbf{M}_{\mathrm{Mma}}$. To summarize, we need the following to imitate Mathematica's behavior:

- For linear matching, from the algorithm $\mathbf{LM}_{\mathrm{S}}$:
  - All common rules, where FVE is replaced by FVE-M.
  - Both free symbol rules, but SVE-F is replaced by SVE, since it may apply to terms with any head, not just to those with free ones.
  - From the commutative rules only Dec-C is needed.
  - From the strict associative rules we need only Dec-A-strict and IVE-A-strict, and they take into account the equation marking, as discussed above.
  - From the strict associative-commutative rules we need only Dec-AC-strict and IVE-AC-strict, and they take into account the equation marking, as discussed above.
- For non-linear matching, from the algorithm $\mathbf{RS}_{\mathrm{S}}$:
  - we need steps 1, 2, 3, 4, and 7, but in step 4, we will not have equations of the form $\overline{x} \approx \tilde{t}[f]$.
  - we do not need the reduction rules.

We denote the obtained algorithm by $\mathbf{M}_{\mathrm{Mma}}$ to indicate that it (tries to) imitate the Mathematica variant of variadic equational matching. Since the latter is incomplete, it is not surprising that to get $\mathbf{M}_{\mathrm{Mma}}$, we had to omit or restrict some rules in our algorithm.

Example 14. We apply $\mathbf{M}_{\mathrm{Mma}}$ to the problem (7): $f(x,y) \ll_E f(a,b)$ with $E = \{\mathsf{A}_{\mathrm{s}}(f)\}$. First alternative, start from Dec-A-strict:

$\{f(x,y) \ll_E f(a,b)\}; \emptyset \rightsquigarrow_{\text{Dec-A-strict}}$
  $f(y) \ll_E f(b)$ gets marked by 0.
$\{x \ll_E a, f(y) \ll_E f(b)\}; \emptyset \rightsquigarrow_{\text{IVE}}$
$\{f(y) \ll_E f(b)\}; \{x \approx a\} \rightsquigarrow_{\text{Dec-A-strict}}$
  Since $f(y) \ll_E f(b)$ is marked by 0, IVE-A-strict cannot apply:
  $f(b)$ has too few arguments to be assigned to y by IVE-A-strict.
$\{y \ll_E b, f() \ll_E f()\}; \{x \approx a\} \rightsquigarrow_{\text{IVE}}$
$\{f() \ll_E f()\}; \{x \approx a, y \approx b\} \rightsquigarrow_{\mathsf{T}}$
$\emptyset; \{x \approx a, y \approx b\}$.

Second alternative, start from IVE-A-strict with $f(a)$:

$\{f(x,y) \ll_E f(a,b)\}; \emptyset \rightsquigarrow_{\text{IVE-A-strict}}$
  $f(y) \ll_E f(b)$ gets marked by 1.
$\{x \ll_E f(a), f(y) \ll_E f(b)\}; \emptyset \rightsquigarrow_{\text{IVE}}$
$\{f(y) \ll_E f(b)\}; \{x \approx f(a)\} \rightsquigarrow_{\text{IVE-A-strict}}$
  Since $f(y) \ll_E f(b)$ is marked by 1, Dec-A-strict cannot apply.
$\{y \ll_E f(b), f() \ll_E f()\}; \{x \approx f(a)\} \rightsquigarrow_{\text{IVE}}$
$\{f() \ll_E f()\}; \{x \approx f(a), y \approx f(b)\} \rightsquigarrow_{\mathsf{T}}$
$\emptyset; \{x \approx f(a), y \approx f(b)\}$.

Third alternative, start from IVE-A-strict with $f(a,b)$:

$\{f(x,y) \ll_E f(a,b)\}; \emptyset \rightsquigarrow_{\text{IVE-A-strict}}$
  $f(y) \ll_E f()$ stays unmarked.
$\{x \ll_E f(a,b), f(y) \ll_E f()\}; \emptyset \rightsquigarrow_{\text{IVE}}$
$\{f(y) \ll_E f()\}; \{x \approx f(a,b)\}$
  No applicable rule. Fail.

Hence, $\mathbf{M}_{\text{Mma}}$ computes two substitutions $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto f(a), y \mapsto f(b)\}$. This is also what Mathematica returns.

Example 15. Let the equation be $g(X(x,y), X(y,x)) \ll_E g(f(a,b), f(b,a))$, where $E = \{\mathsf{C}(f)\}$. Since $\mathbf{M}_{\text{Mma}}$ expects normalized input, we should actually take

$$g(X(x,y), X(y,x)) \ll_E g(f(a,b), f(a,b)).$$

Linearization transforms it into

$$g(X_1(x_1,y_1), X_2(y_2,x_2)) \ll_E g(f(a,b), f(a,b)).$$

By Dec-F, we get two equations

$$X_1(x_1,y_1) \ll_E f(a,b),\ X_2(y_2,x_2) \ll_E f(a,b).$$

The first one gives the solution $\sigma_1 = \{X_1 \mapsto f, x_1 \mapsto a, y_1 \mapsto b\}$. The second one is solved by $\sigma_2 = \{X_2 \mapsto f, x_2 \mapsto b, y_2 \mapsto a\}$. Since $x_1\sigma_1 \neq x_2\sigma_2$, the solutions are inconsistent and, hence, $\mathbf{M}_{\text{Mma}}$ cannot solve the problem. This is also how Mathematica behaves.

At the end of this section, we note that Mathematica has matching with yet another equational theory, called `OneIdentity`. It can be characterized by the axiom $f(x) \approx x$. It is an example of a collapse theory, where a term is equal to its proper subterm (Siekmann, 1989). It implies that the equivalence class of a variable does not consist of that variable only. Such theories need a special treatment and we have not considered it in our work. It can be a subject of future investigations.

8. Discussion and conclusion

We studied matching in variadic equational theories for associativity, commutativity, and their combination. Variadic A-matching and AC-matching in the presence of sequence variables are infinitary. Therefore, any procedure that tries to directly enumerate their complete set of matchers is nonterminating. However, looking at those matchers closer, one can see that there is a pretty regular way to obtain them from finitely many basic ones. This observation allowed us to develop a terminating matching algorithm for the mentioned variadic equational theories. The algorithm computes a finite representation of a possibly infinite complete set of matchers. It works in two

steps: first, the given matching problem is linearized by replacing multiple occurrences of the same variable with fresh variables. Next, the obtained linear problem is solved by an algorithm called **LM**. It gives a finite representation of potentially infinite set of matchers in the form of solved equations of certain kind. Finally, from the solutions of the linear problem we reconstruct solutions of the original one by another algorithm, which is called **RS**. Termination, soundness, and completeness of **LM** and **RS** are proved.

After that, we looked into special matching problems for which the complete set of matchers is finite. It requires certain restrictions on occurrences of sequence variables in the problem or in the solutions. We identified two such cases: a bounded fragment and a strict variant. In the bounded fragment, each sequence variable either is an argument of two different function symbols, or never appears as an argument of any associative and associative-commutative symbol. This condition implies that solutions have a bound on their size and, hence, there are finitely many matchers.

The strictness property imposes a restriction on the way nested associative function symbols are flattened: The inner one should have at least one argument. The strict variant requires us to compute matchers modulo the strictness property. There are finitely many such matchers. Our algorithms need only a minor modification to compute them. We called the algorithms adapted to the strict variant $\mathbf{LM}_{\mathrm{S}}$ and $\mathbf{RS}_{\mathrm{S}}$.

Further, we made an attempt to understand the behavior of the powerful variadic equational matching algorithm of the Mathematica system. Our analysis suggests that it corresponds to an incomplete strict variant of a fragment normalized with respect to the given equational theory, where incompleteness stems from the goal of making it efficient. To model its behavior, we restricted $\mathbf{LM}_{\mathrm{S}}$ and $\mathbf{RS}_{\mathrm{S}}$ and obtained an algorithm which we called $\mathbf{M}_{\mathrm{Mma}}$. Its evaluation showed that for those examples we experimented with, $\mathbf{M}_{\mathrm{Mma}}$ computes the same set of answers as Mathematica's equational (flat, orderless) matching algorithm.

Acknowledgments.

This research has been partially supported by the Austrian Science Fund (FWF) under the project 28789-N32 and the Shota Rustaveli National Science Foundation of Georgia under the grant FR-19-18557.

References

Baader, F., Snyder, W., 2001. Unification theory. In: Robinson, J. A., Voronkov, A. (Eds.), Handbook of Automated Reasoning (in 2 volumes). Elsevier and MIT Press, pp. 445–532.

Barthels, H., Psarras, C., Bientinesi, P., 2019. Automatic generation of efficient linear algebra programs. CoRR abs/1907.02778.
URL http://arxiv.org/abs/1907.02778

Benanav, D., Kapur, D., Narendran, P., 1987. Complexity of matching problems. J. Symb. Comput. 3 (1/2), 203–216.
URL https://doi.org/10.1016/S0747-7171(87)80027-5

Buchberger, B., 1996. Mathematica as a rewrite language. In: Ida, T., Ohori, A., Takeichi, M. (Eds.), Proceeding of the 2nd Fuji International Workshop on Functional and Logic Programming. World Scientific, pp. 1–13.

Buchberger, B., Craciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W., 2006. Theorema: Towards computer-aided mathematical theory exploration. J. Appl. Logic 4 (4), 470–504.
URL https://doi.org/10.1016/j.jal.2005.10.006

Chasseur, E., Deville, Y., 1997. Logic program schemas, constraints, and semi-unification. In: Fuchs (1998), pp. 69–89.
URL https://doi.org/10.1007/3-540-49674-2_4

Cirstea, H., Kirchner, C., Kopetz, R., Moreau, P.-E., 2010. Anti-patterns for rule-based languages. J. Symb. Comput. 45 (5), 523–550.
URL https://doi.org/10.1016/j.jsc.2010.01.007

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. L. (Eds.), 2007. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Vol. 4350 of Lecture Notes in Computer Science. Springer.
URL https://doi.org/10.1007/978-3-540-71999-1

Coelho, J., Dundua, B., Florido, M., Kutsia, T., 2010. A rule-based approach to XML processing and web reasoning. In: Hitzler, P., Lukasiewicz,

T. (Eds.), Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings. Vol. 6333 of Lecture Notes in Computer Science. Springer, pp. 164–172.
URL http://dx.doi.org/10.1007/978-3-642-15918-3_13

Coelho, J., Florido, M., 2004. CLP(Flex): constraint logic programming applied to XML processing. In: Meersman, R., Tari, Z. (Eds.), On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25-29, 2004, Proceedings, Part II. Vol. 3291 of Lecture Notes in Computer Science. Springer, pp. 1098–1112.
URL https://doi.org/10.1007/978-3-540-30469-2_17

Coelho, J., Florido, M., 2007. XCentric: A logic-programming language for XML processing. In: PLAN-X 2007, Programming Language Technologies for XML, An ACM SIGPLAN Workshop colocated with POPL 2007, Nice, France, January 20, 2007. pp. 93–94.
URL http://www.plan-x-2007.org/plan-x-2007.pdf

Dershowitz, N., Manna, Z., 1979. Proving termination with multiset orderings. Commun. ACM 22 (8), 465–476.
URL https://doi.org/10.1145/359138.359142

Dundua, B., 2014. Programming with sequence and context variables: Foundations and applications. Ph.D. thesis, University of Porto.

Dundua, B., Florido, M., Kutsia, T., Marin, M., 2016. CLP(H): constraint logic programming for hedges. TPLP 16 (2), 141–162.
URL https://doi.org/10.1017/S1471068415000071

Dundua, B., Kutsia, T., Marin, M., 2009. Strategies in PρLog. In: Fernández, M. (Ed.), 9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS'09. Vol. 15 of EPTCS. pp. 32–43.
URL https://doi.org/10.4204/EPTCS.15

Dundua, B., Kutsia, T., Marin, M., 2019. Variadic equational matching. In: Kaliszyk, C., Brady, E., Kohlhase, A., Coen, C. S. (Eds.), Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings. Vol. 11617 of Lecture Notes

in Computer Science. Springer, pp. 77–92.
URL https://doi.org/10.1007/978-3-030-23250-4_6

Dundua, B., Kutsia, T., Reisenberger-Hagmayer, K., 2017. An overview of PρLog. In: Lierler, Y., Taha, W. (Eds.), Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings. Vol. 10137 of Lecture Notes in Computer Science. Springer, pp. 34–49.
URL https://doi.org/10.1007/978-3-319-51676-9_3

Eker, S., 2002. Single elementary associative-commutative matching. J. Autom. Reasoning 28 (1), 35–51.
URL https://doi.org/10.1023/A:1020122610698

Eker, S., 2003. Associative-commutative rewriting on large terms. In: Nieuwenhuis (2003), pp. 14–29.
URL https://doi.org/10.1007/3-540-44881-0_3

Fages, F., Huet, G. P., 1986. Complete sets of unifiers and matchers in equational theories. Theor. Comput. Sci. 43, 189–200.
URL https://doi.org/10.1016/0304-3975(86)90175-1

Fuchs, N. E. (Ed.), 1998. Logic Programming Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 10-12, 1997, Proceedings. Vol. 1463 of Lecture Notes in Computer Science. Springer.
URL https://doi.org/10.1007/3-540-49674-2

Hamana, M., 1997. Term rewriting with sequences. In: Proceedings of the First International Theorema Workshop. No. 97-20 in RISC Technical Report series. Hagenberg, Austria.

Horozal, F., Rabe, F., Kohlhase, M., 2014. Flexary operators for formalized mathematics. In: Watt, S. M., Davenport, J. H., Sexton, A. P., Sojka, P., Urban, J. (Eds.), Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings. Vol. 8543 of Lecture Notes in Computer Science. Springer, pp. 312–327.
URL https://doi.org/10.1007/978-3-319-08434-3_23

Horozal, F. F., 2014. A framework for defining declarative languages. Ph.D. thesis, Jacobs University Bremen.

Hosoya, H., Pierce, B. C., 2003. Regular expression pattern matching for XML. J. Funct. Program. 13 (6), 961–1004.
URL https://doi.org/10.1017/S0956796802004410

Hullot, J., 1979. Associative commutative pattern matching. In: Buchanan, B. G. (Ed.), Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes. William Kaufmann, pp. 406–412.
URL http://ijcai.org/proceedings/1979-1

International Organization for Standardization, 2018. Information technology — Common Logic (CL) — a framework for a family of logic-based languages. International Standard ISO/IEC 24707:2018(E). Available online at https://www.iso.org/standard/66249.html.

Krebber, M., Barthels, H., Bientinesi, P., 2017. Efficient pattern matching in Python. In: Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing, PyHPC@SC 2017, Denver, CO, USA, November 12, 2017. ACM, pp. 2:1–2:9.
URL https://doi.org/10.1145/3149869.3149871

Kutsia, T., 2002a. Solving and proving in equational theories with sequence variables and flexible arity symbols. RISC Report Series 02-09, RISC, Johannes Kepler University Linz.

Kutsia, T., 2002b. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In: Calmet, J., Benhamou, B., Caprotti, O., Henocque, L., Sorge, V. (Eds.), Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Joint International Conferences, AISC 2002 and Calculemus 2002, Marseille, France, July 1-5, 2002, Proceedings. Vol. 2385 of Lecture Notes in Computer Science. Springer, pp. 290–304.
URL https://doi.org/10.1007/3-540-45470-5_26

Kutsia, T., 2003. Equational prover of Theorema. In: Nieuwenhuis (2003), pp. 367–379.
URL https://doi.org/10.1007/3-540-44881-0_26

Kutsia, T., 2004. Solving equations involving sequence variables and sequence functions. In: Buchberger, B., Campbell, J. A. (Eds.), Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC

2004, Linz, Austria, September 22-24, 2004, Proceedings. Vol. 3249 of Lecture Notes in Computer Science. Springer, pp. 157–170.
URL https://doi.org/10.1007/978-3-540-30210-0_14

Kutsia, T., 2007. Solving equations with sequence variables and sequence functions. J. Symb. Comput. 42 (3), 352–388.
URL https://doi.org/10.1016/j.jsc.2006.12.002

Kutsia, T., 2008. Flat matching. J. Symb. Comput. 43 (12), 858–873.
URL https://doi.org/10.1016/j.jsc.2008.05.001

Kutsia, T., Marin, M., 2005. Can context sequence matching be used for querying XML? In: Vigneron, L. (Ed.), Proceedings of the 19th International Workshop on Unification (UNIF'05). Nara, Japan, pp. 77–92.

Kutsia, T., Marin, M., 2012. Solving, reasoning, and programming in Common Logic. In: Voronkov, A., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S. M., Zaharie, D. (Eds.), 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012. IEEE Computer Society, pp. 119–126.
URL https://doi.org/10.1109/SYNASC.2012.27

Kutsia, T., Marin, M., 2015. Regular expression order-sorted unification and matching. J. Symb. Comput. 67, 42–67.
URL https://doi.org/10.1016/j.jsc.2014.08.002

Marin, M., Kutsia, T., 2003. On the implementation of a rule-based programming system and some of its applications. In: Konev, B., Schmidt, R. (Eds.), Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03). Almaty, Kazakhstan, pp. 55–68.

Marin, M., Kutsia, T., 2006. Foundations of the rule-based system ρLog. Journal of Applied Non-Classical Logics 16 (1-2), 151–168.
URL https://doi.org/10.3166/jancl.16.151-168

Marin, M., Tepeneu, D., 2003. Programming with sequence variables: The Sequentica package. In: Challenging The Boundaries Of Symbolic Computation. Proc. 5th Int. Mathematica Symposium. World Scientific, pp. 17–24.

Nieuwenhuis, R. (Ed.), 2003. Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings. Vol. 2706 of Lecture Notes in Computer Science. Springer.
URL https://doi.org/10.1007/3-540-44881-0

Pease, A., Sutcliffe, G., 2007. First order reasoning on a large ontology. In: Sutcliffe, G., Schulz, S. (Eds.), Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories. No. 257 in CEUR Workshop Proceedings. pp. 59–69.

Richardson, J., Fuchs, N. E., 1997. Development of correct transformation schemata for Prolog programs. In: Fuchs (1998), pp. 263–281.
URL https://doi.org/10.1007/3-540-49674-2

Schaefer, T. J., 1978. The complexity of satisfiability problems. In: Lipton, R. J., Burkhard, W. A., Savitch, W. J., Friedman, E. P., Aho, A. V. (Eds.), Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA. ACM, pp. 216–226.
URL https://doi.org/10.1145/800133.804350

Siekmann, J. H., 1989. Unification theory. J. Symb. Comput. 7 (3/4), 207–274.
URL https://doi.org/10.1016/S0747-7171(89)80012-4

Trott, M., 2004. The Mathematica guidebook for programming (includes DVD). Springer.
URL http://www.worldcat.org/oclc/178804217

Wolfram, S., 2003. The Mathematica book, 5th Edition. Wolfram-Media.