Computing in Theories Defined by Permutative Equations

Jürgen Avenhaus
FB Informatik
Universität Kaiserslautern

November 5, 2003

Abstract

In automated deduction it is sometimes helpful to compute modulo a set E of equations. In this paper we consider the case where E consists of permutative equations only. In this case it is not easy to represent any E-equivalence class $[t]_E$ by just one representative and to compute with these representatives efficiently. Therefore, in [AP01] a method is presented to modify E to $\hat{E}$ and to approximate the E-equivalence classes $[t]_E$ by $\hat{E}$-equivalence classes $[t]_{\hat{E}}$ for "stratified" terms t. Then one can compute relatively efficiently with these $\hat{E}$-equivalence classes. In this paper we improve algorithms given in [AP01]. Provided that E satisfies some conditions, one can avoid the construction of [AP01] and compute with the E-equivalence class directly. Our results apply to this case, too.
If E is allowed to be part of the input then even testing $\hat{E}$-equality is at least as hard as testing for graph isomorphism. For a fixed set E we present a polynomial time algorithm for testing $\hat{E}$-equality. Testing matchability and unifiability modulo $\hat{E}$ is NP-complete. The same holds for $\hat{E}$ replaced by E. We present relatively efficient algorithms for these problems. Our algorithms are based on knowledge from group theory.

1 Introduction

1.1 Overview on the paper

In automated deduction one is concerned with the following problem: Given a signature sig and a set $\mathcal{A}x$ of axioms, try to prove that a closed formula F follows from $\mathcal{A}x$ or – more precisely – is valid in all models of the specification spec $= (\text{sig}, \mathcal{A}x)$. For this problem powerful theorem provers have been developed over the past years. The provers work on clauses. See [BG01] for a recent overview on resolution methods. If $\mathcal{A}x$ contains an "appropriate" set E of universally closed equations then it may be of considerable advantage to compute modulo E, i.e. to work on clauses C that represent the whole E-equivalence class of C (or a large part of it). The most prominent example is that E is a set of AC-axioms (associativity and commutativity) of some binary function symbols [NR01]. In this paper we assume that E is a set of permutative equations [AP01].

The most basic algorithmic problems for computing modulo E are testing for equality, matching and unification modulo E. We will present algorithms for these problems that are based on knowledge about permutation groups.

To give a simple example, assume that E is given by the permutation axioms

$$E = \{f(\bar{x}) = f(\pi_i(\bar{x})) \mid i = 1, \ldots, n\}$$

where $\bar{x} = (x_1, \ldots, x_k)$ is a vector, the π_i are permutations on $\mathbb{N}_k = \{1, \ldots, k\}$ and $\pi(\bar{x}) = \pi(x_1, \ldots, x_k) = (x_{\pi(1)}, \ldots, x_{\pi(k)})$ for any permutation π. Let $G = \langle \pi_1, \ldots, \pi_n \rangle$ be the permutation group generated by $\pi_1, \ldots, \pi_n$. Then we have for f-free terms s_i and t_j and $s \equiv f(s_1, \ldots, s_k)$ and $t \equiv f(t_1, \ldots, t_k)$: $s =_E t$ iff $\pi(\bar{s}) = \bar{t}$ for some $\pi \in G$. It turns out that even this problem is at least as hard as testing whether two graphs with k vertices are isomorphic. On the other hand, if the s_i are pairwise distinct, then this problem reduces to test for a given permutation π whether π is in G. This can be done in polynomial time [FHL80]. After a preprocessing of the $\pi_1, \ldots, \pi_n$ this can be done in time $O(k^2)$.

Now assume that the arities of all top symbols of equations in E are bounded by some $k \in \mathbb{N}$. Let $n = |t|$ be the length of the term t. Based on knowledge about permutation groups we present efficient algorithms for the problems equality, matching and unification modulo E. For E fixed, the test for E-equality is polynomial in $n = |s| + |t|$.

The results can be extended to the case that E is a set of permutative equations. That is, E is the union of some E_c and $E_c = \{c[\bar{x}] = c[\pi_i(\bar{x})] \mid i = 1, \ldots, n\}$. Here $c[\Box_1, \ldots, \Box_n]$ is a context (i.e., a term that contains each $\Box_i$ exactly once). In this case it is not so easy to represent the whole E-equivalence class $[t]_E$ by one term. Instead, we follow [AP01] and use a "stratified" version $\hat{t}$ of t to represent by $\hat{t}$ the subclass $um[\hat{t}]_E \subseteq [t]_E$. We describe algorithms for testing equality, matchability and unifiability for this approximation for computing modulo E.

This paper is not on the theory of unification. See [BS01] and [JK91] for surveys on unification theory. The emphasis of the paper is to present methods to enhance the efficiency of known algorithms. These methods are based on knowledge from the theory of permutation groups from the theoretical point of view and on our experience from writing efficient theorem provers such as WaldMeister [HL02] from the implementation point of view. From this experience we learned that it is more efficient to use simple algorithms based on simple data structures than to use sophisticated algorithms based on complex data structures (e.g. multisets as in [AP01]) to avoid redundancies in search problems. This does not prevent usage of sharing and memoization techniques to avoid solving the same subproblem repeatedly. Knowledge about permutation groups allows us to enumerate the permutations $\pi \in G$ in an intelligent way (e.g. to detect failure early) and the implementation techniques allow us to reuse already computed information efficiently. For complexity results on related problems see [dlTE03].

We agree that for simple problems it is more efficient to use the naive algorithms for testing equality, matchability and unifiability modulo E. But if deep terms and large permutation groups occur our algorithms will outperform the naive ones.

The paper is organized as follows. In section 2 we present what is needed from group theory later on. Section 3 is devoted to the case that E is a set of permutation equations. Here we present our methods in detail. In section 4 it is outlined how to extend these methods to the case where E is a set of permutative equations.

1.2 Notations

We use standard notations. A (one sorted) signature is a pair $\mathrm{sig} = (F, \alpha)$ where F is a set of function symbols and $\alpha : F \to \mathbb{N}$ fixes the arity of any $f \in F$. Let V be a set of variables, then $\mathrm{Term}(F, V)$ is the set of terms over F and V. We write $s \equiv t$ if s and t are syntactically equal. The length of t is denoted by $|t|$ and $\mathrm{depth}(t)$ is its depth. The top symbol of a term t is denoted by $\mathrm{top}(t)$, i.e. $\mathrm{top}(t) = f$ if $t \equiv f(t_1, \ldots, t_n)$ and $\mathrm{top}(t) = x$ if $t \equiv x$. $\mathrm{Var}(t)$ is the set of variables in t and $\mathrm{Sub}(t)$ is the set of all subterms of t. We denote by t/p the subterm of t at position p and write $t[s]_p$ to denote t with the subterm t/p replaced by s. An equation has the form $s = t$ with $s, t \in \mathrm{Term}(F, V)$.

A substitution is $\sigma : V \to \mathrm{Term}(F, V)$ such that the domain $\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \not\equiv x\}$ is finite. Then σ is extended to $\sigma : \mathrm{Term}(F, V) \to \mathrm{Term}(F, V)$ homomorphically as usual. σ is a *match* from s to t iff $\sigma(s) \equiv t$. A *unifier* of s and t is a substitution σ such that $\sigma(s) \equiv \sigma(t)$. As usual we generalize the concept of unifiers to sets E of equations: σ is a unifier for E if $\sigma(s) \equiv \sigma(t)$ for all $s = t$ in E. If there is a unifier of E then there always exists a *most general unifier* $\sigma = \mathrm{mgu}(E)$.

Let E be a set of equations. Then $t_1 \vdash\!\dashv_E t_2$ iff $t_1 \equiv t_1[\sigma(u)]_p$ and $t_2 \equiv t_1[\sigma(v)]_p$ for some $u = v$ in E and some σ. Then $=_E$ is the reflexive, symmetric and transitive closure of $\vdash\!\dashv_E$. Two terms s, t are *E-equal* iff $s =_E t$.

A substitution σ is an *E-match* from s to t iff $\sigma(s) =_E t$. And σ is an *E-unifier* of s and t if $\sigma(s) =_E \sigma(t)$. We write $\sigma_1 =_E \sigma_2$ iff $\sigma_1(x) =_E \sigma_2(x)$ for all $x \in \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)$. A set Σ of substitutions is a *complete set of unifiers* of s, t (a CSU_E for s and t) iff for each unifier τ of s, t there is a $\sigma \in \Sigma$ and a σ' such that $\tau =_E \sigma' \circ \sigma$. We sometimes write $\mathrm{CSU}_E(s, t)$ to denote any CSU_E for s, t.

For doing rewriting or deduction modulo E (see e.g. [BN98], [Ave95], [NR01]) one needs to solve the following problems: For given $s, t \in \mathrm{Term}(F, V)$ decide whether

(1) s is E-equal to t,

(2) s is E-matchable to t,

(3) s is E-unifiable with t.

For problem (3) it may be necessary to compute a $\mathrm{CSU}_E(s, t)$. In the rest of the paper we will study these problems for a set E of permutative equations.

2 Permutative equivalence on strings

In this section we consider two problems on strings of a fixed length. These problems (and their solution) are basic to the rest of the paper. We first recall some results on permutation groups to solve these problems.

A *permutation* on $\mathbb{N}_k = \{1, \dots, k\}$ is a bijective function $\pi : \mathbb{N}_k \to \mathbb{N}_k$. These permutations form a group $\mathrm{Sym}(k)$ by using composition of functions as the group multiplication and the inverse function π^{-1} of as the group inverse. Then id (the identity mapping) is the unit element of $\mathrm{Sym}(k)$. For $\pi_1, \dots, \pi_n \in \mathrm{Sym}(k)$ we denote by $G = \langle \pi_1, \dots, \pi_n \rangle$ the subgroup of $\mathrm{Sym}(k)$ generated by the π_i: G consists of all finite products of the π_i and π_i^{-1}.

Theorem 2.1 ([FHL80])
The following problem is solvable in polynomial time.
Input: $\pi_1, \dots, \pi_n, \pi \in \mathrm{Sym}(k)$
Question: Is π in $G = \langle \pi_1, \dots, \pi_n \rangle$? □

We outline a proof of this theorem because it forms the basis for the algorithms in the next sections.
From group theory we know that any $\pi \in G$ is a product
$\pi = g_1 \circ \dots \circ g_k$ with

$g_i(m) = m \qquad \text{for } 1 \le m < i \le k$

From this it follows that

$$\begin{aligned} \pi(i) &= g_1 \cdots g_k(i) = g_1 \cdots g_i(i) && \text{for } i = 1, \ldots, k \\ g_i(i) &= g_{i-1}^{-1} \cdots g_1^{-1} \pi(i) && \text{for } i = 1, \ldots, k \end{aligned}$$

So π can be "approximated" by computing $g_1, \ldots, g_k$.

The algorithm that proves the theorem tries to approximate all $\pi \in G$ simultaneously. To do so it computes from $\pi_1, \ldots, \pi_n$ permutations $g_{ij} \in G$ for $1 \le i \le j \le k$ such that

$$\begin{aligned} g_{ij}(m) &= m && \text{for } 1 \le m < i \\ g_{ij}(m) &= j && \text{for } i = m \end{aligned}$$

and stores g_{ij} in a table T at position (i, j). If no such $g_{ij} \in G$ exists then the table entry at position (i, j) is empty. (See [FHL80] for details how to do this.) We may choose $g_{ii} = id$. Provided the table T is already computed, the following algorithm can be used to decide for any $\pi \in \mathrm{Sym}(k)$ whether $\pi \in G$ holds and if so to compute the canonical form (relative to T) $\pi = g_{1j_1} \circ \cdots \circ g_{kj_k}$ of π:

for $i = 1$ **to** k **do**
  $j := j_i := \pi(i)$;
  if position(i, j) of T is empty
    then STOP with NO
    else $\pi := g_{ij}^{-1} \circ \pi$
od
STOP with YES and output $\pi = g_{1j_1} \circ \cdots \circ g_{kj_k}$

Example 2.1
Let $k = 6$ and π_1, π_2 be given by

i	1	2	3	4	5	6
π_1	2	3	4	5	6	1
π_2	5	4	3	2	1	6

This results in T with non-empty entries at positions (i, i) and $(1, j)$ for $j = 2, \ldots, 6$ and $(2, 6)$ only:

$T:$	id	g_{12}	g_{13}	g_{14}	g_{15}	g_{16}
		id	$-$	$-$	$-$	g_{26}
			id	$-$	$-$	$-$
				id	$-$	$-$
					id	$-$
						id

The g_{ij} are given by

	1	2	3	4	5	6
$g_{12} = \pi_1$	2	3	4	5	6	1
$g_{13} = \pi_1^2$	3	4	5	6	1	2
$g_{14} = \pi_1^3$	4	5	6	1	2	3
$g_{15} = \pi_2$	5	4	3	2	1	6
$g_{16} = \pi_1\pi_2$	6	5	4	3	2	1
$g_{26} = \pi_1^2\pi_2$	1	6	5	4	3	2

As a result, each $\pi \in G$ has the canonical form $\pi = g_{1j}g_{2m}$ with $1 \leq j \leq 6$ and $m \in \{2, 6\}$. So G has exactly $6 \times 2 = 12$ elements.

$\square$

We remark that the algorithm to compute T is non-deterministic, so for input $\pi_1, \ldots, \pi_n$ several tables T may exist. Hence, the canonical form of $\pi \in G$ is unique with respect to T only, not with respect to $\pi_1, \ldots, \pi_n$. As a second remark we notice that the computation of T in Example 2.1 was done according to the sequence $(1, 2, \ldots, k)$. One may choose any sequence $(\varrho(1), \ldots, \varrho(k))$, $\varrho \in \mathrm{Sym}(k)$, to compute a table T with entries at positions $(\varrho(i), \varrho(j))$ such that $g_{\varrho(i)\varrho(i)} = id$ and $g_{\varrho(i)\varrho(j)}(\varrho(m)) = \varrho(m)$ for $1 \leq m \leq i-1$ and $g_{\varrho(i)\varrho(i)}(\varrho(i)) = \varrho(j)$.

Now we come to the permutative equivalence problems on strings. We use the following notation. For any alphabet $A = \{a_1, \ldots, a_r\}$, $u = u_1 \cdots u_k \in A^k$ and $\pi \in \mathrm{Sym}(k)$ we write $\pi(u) = u_{\pi(1)} \cdots u_{\pi(k)}$.

Definition 2.1
The *permutative equivalence problem* (PEP) for strings is
Input: $G = \langle \pi_1, \cdots, \pi_n \rangle$ a subgroup of $\mathrm{Sym}(k)$
$u, v \in A^k$.
Question: Is there a $\pi \in G$ such that $\pi(u) = v$?
The *specialized permutative equivalence problem* (SPEP) is the same as PEP but with the restriction that u contains any $a \in A$ at most once.

Lemma 2.1
The problem SPEP is solvable in polynomial time.

Proof: Test whether there is a $\pi \in \mathrm{Sym}(k)$ such that $u = \pi(v)$. (There exists at most one π.) Now test whether $\pi \in G$ holds. $\square$

Now we show that PEP is much harder than SPEP by proving that the isomorphism problem for graphs is polynomial time reducible to PEP. This reduction was suggested to me by Peter Cameron [Cam02] and by David Plaisted [Pla95].

A *k-graph* is $\mathcal{G} = (V, E)$ with $\|V\| = k$ and $E \subseteq V \times V$. We may assume that $V = \mathbb{N}_k = \{1, \ldots, k\}$. $\mathcal{G}$ is *isomorphic* to $\mathcal{G}' = (\mathbb{N}_k, E')$ if there is a $\pi \in \mathrm{Sym}(k)$ such that $\pi(E) = E'$. Here $\pi(E)$ denotes the set $\pi(E) = \{(\pi(i), \pi(j)) \mid (i, j) \in E\}$. Each set $E \subseteq \mathbb{N}_k \times \mathbb{N}_k$ can be uniquely represented by its "characteristic string" $u_E \in \{0, 1\}^n$ with $n = k^2$: Let $\varphi : \mathbb{N}_k \times \mathbb{N}_k \to \mathbb{N}_n$ by any bijection.
Define $u_E = u_1 \cdots u_n$ by

$$u_{\varphi(i,j)} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases}$$

For any $E_1, E_2 \subseteq \mathbb{N}_k \times \mathbb{N}_k$ we have $E_1 = E_2$ iff $u_{E_1} = u_{E_2}$. Now we define for any $\pi \in \mathrm{Sym}(k)$ a $\pi^* \in \mathrm{Sym}(n)$ such that $\pi^*(u_E) = u_{\pi(E)}$. This π^* is given by

$$\pi^*(\varphi(i, j)) = \varphi(\pi(i), \pi(j))$$

Let G be the subgroup of $\mathrm{Sym}(n)$

$$G = \langle \pi^* \mid \pi \in \mathrm{Sym}(k) \rangle$$

Lemma 2.2
Two k-graphs $\mathcal{G} = (\mathbb{N}_k, E)$ and $\mathcal{G}' = (\mathbb{N}_k, E')$ are isomorphic iff there is $\pi^* \in G$ such that $\pi^*(u_E) = u_{E'}$.

Proof: We have indeed $\pi^*(u_E) = u_{\pi(E)}$ for any $\pi \in \mathrm{Sym}(k)$ by definition of u_E and π^*. Now we have that $\mathcal{G}$ and $\mathcal{G}'$ are isomorphic iff $\pi(E) = E'$ for some $\pi \in \mathrm{Sym}(k)$ iff $\pi^*(u_E) = u_{E'}$ for some $\pi^* \in G$. $\square$

This Lemma immediately leads to

Theorem 2.2
The graph isomorphism problem GI is polynomial time reducible to PEP. $\square$

It is not known whether GI is NP-complete. But it is widely believed that GI is not solvable in polynomial time. So we do not expect to find a polynomial time algorithm for PEP.
On the other hand, Theorem 2.1 opens the door to an algorithm that works efficiently in many situations: If $u, v \in A^k$ are given and the table T is already computed then there may exist several $\pi \in \mathrm{Sym}(k)$ such that $\pi(u) = v$. We use T to find such a $\pi \in G$ by backtracking. We outline an algorithm how to do this. For any $w \in A^k$ let w_i denote the i-th letter of w. Remember that for any $\pi \in \mathrm{Sym}(k)$ we have $\pi(w) = w_{\pi(1)} \cdots w_{\pi(k)}$, so $\pi(w)_i = w_{\pi(i)}$.

Now assume that $\pi(u) = v$ for some $\pi \in G$. Then we have $\pi = g_1 \circ \cdots \circ g_k$ with g_i in row i of T and so

$$u = g_k^{-1} \cdots g_1^{-1}(v)$$

For computing $g_1, \ldots, g_k$ from u and v notice that $u_i = u_{g_{i+1}\cdots g_k(i)} = g_{i+1}\cdots g_k(u)_i = g_i^{-1}\cdots g_1^{-1}(v)_i$, hence

$$u_i = g_i^{-1}(v')_i \qquad \text{where } v' = g_{i-1}^{-1}\cdots g_1^{-1}(v)$$

So there is a position j in v' with $i \leq j \leq k$ such that $u_i = v'_j$ and $g_i^{-1}(j) = i$. This gives $g_i(i) = j$ and hence $g_i = g_{ij}$ in T. Computing $g_1, \ldots, g_k$ this way leads to the loop invariant that u and $g_i^{-1}\cdots g_1^{-1}(v)$ have a common prefix of length i.

This leads to a non-deterministic algorithm to test whether there is a $\pi \in G$ such that $\pi(u) = v$:

$v' := v$
for $i = 1$ **to** k **do**
    if there is no g_i in row i of T such that $g_i(i) = j$ and $v'_j = u_i$
      then STOP with NO
       else choose such a g_i and put $v' := g_i^{-1}(v')$
od
STOP with YES and $\pi = g_1 \circ \cdots \circ g_k$

Example 2.2 (Example 2.1 continued)

Assume $u = a\ a\ a\ b\ c\ b \qquad v = a\ b\ c\ a\ b\ a$

The first letter of u is a. There are three g_i such that g_i^{-1} moves an a in v to the first place

(1) $g_1 = id$ gives $g_1^{-1}(v) = a\ b\ c\ a\ b\ a$

(2) $g_1 = g_{14}$ gives $g_1^{-1}(v) = a\ b\ a\ a\ b\ c$

(3) $g_1 = g_{16}$ gives $g_1^{-1}(v) = a\ b\ a\ c\ b\ a$

There are two permutations in row 2 of T : id and g_{26}. The second letter of u is a.
In case (1) only g_{26} moves an a in $g_1^{-1}(v)$ to place 2.

(1.1) $g_2 = g_{26}$ gives $g_2^{-1}g_1^{-1}(v) = a\ a\ b\ a\ c\ b$.

In case (2) no g_2^{-1} moves an a in $g_1^{-1}(v)$ to place 2.
In case (3) only g_{26}^{-1} moves an a in $g_1^{-1}(v)$ to place 2.

(3.1) $g_2 = g_{26}$ gives $g_2^{-1}g_1^{-1}(v) = a\ a\ b\ c\ a\ b$.

For $i = 3, \ldots, 6$ we can only choose $g_i = id$. The third letter of u is a and $id^{-1} = id$ does not move the third a in $g_2^{-1}g_1^{-1}(v)$ to place 3. So there is no $\pi \in G$ such that $\pi^{-1}(v) = u$ or, equivalently, $\pi(u) = v$.
The full search tree is as follows

$$\begin{array}{ccc} & v = a\,b\,c\,a\,b\,a & \\ \text{id} & g_{14} & g_{16} \\ a\,b\,c\,a\,b\,a & a\,b\,a\,a\,b\,c & a\,b\,a\,c\,b\,a \\ g_{26} & & g_{26} \\ a\,a\,b\,a\,c\,b & & a\,a\,b\,c\,a\,b \end{array}$$

So there are exactly two strings of the form $a\,a\,u'$ that are permutative equivalent (modulo G) to v, and there is no such string of the form $a\,a\,a\,u''$. □

To implement this algorithm it would be favorable to push the branching points in the search tree as far down as possible. For the example above that means to move in the first step the letter c to the right position in u, i.e. to position 5. Then to move the two b's and then the three a' to the right place. This can be done by computing the table T according to the sequence $(5, 4, 6, 1, 2, 3)$.

In the next section we need to consider a slightly more general problem than PEP. To describe that let $Compat(u, v, \pi)$ be a (compatability) predicate on $A^k \times A^k \times \mathrm{Sym}(k)$. We assume that *Compat* is easy to decide.

Definition 2.2
The *generalized permutative equivalence problem* GPEP is
Input: $G = \langle \pi_1, \ldots, \pi_n \rangle$, a subgroup of $\mathrm{Sym}(k)$
$u, v \in A^k$
$Compat(u, v, \pi)$, a predicate on $A^k \times A^k \times \mathrm{Sym}(k)$.
Question: Is there a $\pi \in G$ such that $Compat(u, v, \pi)$ and $\pi(u) = v$?

It is easy to design an algorithm $\mathcal{A}$ for GPEP in the same way we have done it for PEP. We will use this algorithm $\mathcal{A}$ in the next sections.

3 Permutation axioms

In this section E is a fixed finite set of permutation equations. We will show that for any such fixed E we can decide (1) in deterministic polynomial time (measured in $n = |s| + |t|$) whether $s =_E t$ holds and (2) in non-deterministic polynomial time whether s is E-matchable to t and (3) whether s and t are E-unifiable. Furthermore, we are going to develop fairly efficient algorithms for these problems.

We start with some notations. A *permutation equation* for $f \in F$ is of the form

$$f(x_1, \ldots, x_k) = f(x_{\pi(1)}, \ldots, x_{\pi(k)})$$

for some $\pi \in \mathrm{Sym}(k)$. Let E_f be a set of such permutation equations and let $G_f = \langle \pi \mid f(\bar{x}) = f(\pi(\bar{x})) \text{ is in } E_f \rangle$.

In this section we assume

$$E = \bigcup_{f \in F_0} E_f \qquad F_0 \subseteq F$$

For $f \in F - F_0$ let $G_f = \langle id \rangle$, the identity group.

We want to emphasize that for the time bounds the set E is assumed to be fixed. If E is allowed to be part of the input then the graph isomorphism problem is polynomial time reducible to testing E-equality for permutation axioms E. See the end of this section. But this reduction assumes that function symbols of arbitrary arity are allowed. In our setting the size of E and of all G_f are bounded by a constant independent of $n = |s| + |t|$.

We represent a term t by its maximally shared graph, i.e. by $\mathrm{graph}(t) = (V, E)$ where $V = \mathrm{Sub}(t)$ is the set of vertices and $E = \{(s, s_i) \mid s \equiv f(s_1, \ldots, s_k), i = 1, \ldots, k\}$ is the set of edges. The edge (s, s_i) is labeled by i. In node $s \in \mathrm{Sub}(t)$ we also store $|s|$ and $\mathrm{depth}(s)$. Recall that $\mathrm{Sub}(t)$ is the set of all subterms of t.

We now come to some basic algorithms for computing modulo E. In all our algorithms we will assume that the table T from section 2 is already computed (in polynomial time).

We start with testing E-equality. This is based on the next lemma.

Lemma 3.1

a) $f(s_1, \ldots, s_k) =_E f(t_1, \ldots, t_k)$ iff
   $s_{\pi(i)} =_E t_i$ for some $\pi \in G_f$ and all $i = 1, \ldots, k$.

b) If $s =_E t$ then $\mathrm{top}(s) = \mathrm{top}(t)$ and $|s| = |t|$ and $\mathrm{depth}(s) = \mathrm{depth}(t)$.

Proof: By induction on $\mathrm{depth}(t)$.

Based on this lemma we give a recursive algorithm for testing E-equality that stores in $Eq(s', t')$ for some $s' \in \mathrm{Sub}(s)$ and $t' \in \mathrm{Sub}(t)$ the result of the test "$s' =_E t'$?". We want to avoid repetitions of this test for fixed s', t'. This is supported by representing s and t by the maximally shared graph for the term $*(s, t)$, where $*$ is a new function symbol. But this alone does not exclude repetitions of this test, especially if $ar(f) > 2$ and G_f is large for some $f \in F$. So we store in $\mathrm{Treated}(s', t')$ whether this test is already performed. We use the following predicate (for compatibility)

EqCompat(s, t, π) iff
For all $i = 1, \ldots, ar(f)$: $\mathrm{top}(s_{\pi(i)}) = \mathrm{top}(t_i)$, $|s_{\pi(i)}| = |t_i|$ and
$\mathrm{depth}(s_{\pi(i)}) = \mathrm{depth}(t_i)$.

So for $s \equiv f(s_1, \ldots, s_k)$, $t \equiv f(t_1, \ldots, t_k)$, if *EqCompat*$(s, t, \pi)$ does not hold, then $s_{\pi(i)} =_E t_i$ does not hold for some i.

Algorithm 3.1
Eq-Test

Input: $s, t \in \mathrm{Term}(F, V)$
Output: If $s =_E t$ then YES else NO

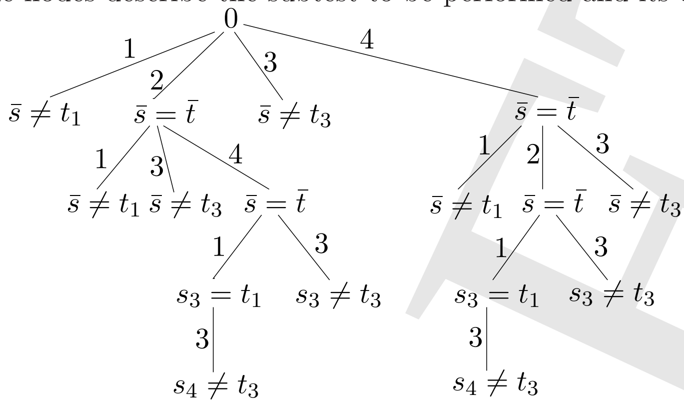
Eq-Test$(s,t) =$
<u>for</u> all $(s', t') \in \mathrm{Sub}(s) \times \mathrm{Sub}(t)$ **<u>do</u>** $\mathrm{Treated}(s', t') :=$ false
TestEq(s,t)
OUTPUT: *Eq*(s,t)

TestEq
TestEq$(s,t) =$
Eq$(s,t) :=$ false
<u>if</u> $\mathrm{top}(s) = \mathrm{top}(t)$ and $|s| = |t|$ and $depth(s) = depth(t)$
  <u>then</u>
    <u>if</u> $s \in V \cup F$
      <u>then</u> $Eq(s,t) := true$
      <u>else</u>
        let $s \equiv f(s_1, \ldots, s_k)$ and $t \equiv f(t_1, \ldots, t_k)$
      <u>for</u> all $\pi \in G_f$ such that $EqCompat(s, t, \pi)$ **<u>do</u>**
        $ok\pi :=$ true; $i := 1$
        <u>while</u> $i \leq k$ and $ok\pi$ **<u>do</u>**
          <u>if</u> not $\mathrm{Treated}(s_{\pi(i)}, t_i)$ **<u>then</u>** $TestEq(s_{\pi(i)}, t_i)$ **<u>fi</u>**
          <u>if</u> $Eq(s_{\pi(i)}, t_i) =$ false **<u>then</u>** $ok\pi :=$ false **<u>fi</u>**
        <u>od</u>
        <u>if</u> $ok\pi =$ true **<u>then</u>** $\{Eq(s,t) :=$ true; $Treated(s,t) :=$ true; $exit\}$ **<u>fi</u>**
      <u>od</u>
    <u>fi</u>
<u>fi</u>
$Treated(s,t) :=$ true

Notice that the enumeration of all $\pi \in G_f$ such that $EqCompat(s, t, \pi)$ is supported by algorithm $\mathcal{A}$ from Section 2. This takes also into account for which i the π-loop was interrupted. We demonstrate this by an example.

Example 3.1
Assume $E = \{f(x_1, x_2, x_3, x_4) = f(x_2, x_3, x_4, x_1), f(x_1, x_2, x_3, x_4) = f(x_2, x_1, x_3, x_4)\}$.
Then $G_f = \mathrm{Sym}(4)$ is the full permutation group on $\mathbb{N}_4$.
Question: Is $s \equiv f(s_1, \ldots, s_4) =_E f(t_1, \ldots, t_4) \equiv t$?
The naive algorithm generates all 4!=24 permutations $\pi \in G_f$ and tests whether $s_{\pi(i)} =_E t_i$ holds for all $i = 1, \ldots, 4$. This results in $4 \cdot 24 = 96$ subtests of the form "$s_i =_E t_j$?". Note that these tests may be expensive.

Now assume $s_1 \equiv s_2 \equiv \bar{s}$, $t_2 \equiv t_4 \equiv \bar{t}$ and $\bar{s} =_E \bar{t}$, $\bar{s} \neq_E t_1$, $\bar{s} \neq_E t_3$, $s_3 =_E t_1$, $s_3 \neq_E t_3$ and $s_4 \neq_E t_3$. We describe the computation of *TestEq* for this input by a tree. Here the numbers near the edges at level i denote the possible values of $\pi(i)$. The nodes describe the subtest to be performed and its outcome.



So $TestEq(s,t)$ calls itself with input (s_i, t_j) only six times (as mentioned in the assumption on the input) compared to 96 calls of the naive algorithm. The outcome of the computation is $f(s_1, \ldots, s_4) \neq_E f(t_1, \ldots, t_4)$. $\square$

Theorem 3.1
Let E be a fixed set of permutation equations and T its table as in section 2. Algorithm *Eq-Test*=$\textit{Eq-Test}(E,T,s,t)$ is correct. It runs in time $O(n^2)$, where $n = |s| + |t|$.

Proof: By Lemma 3.1 the algorithm *TestEq* satisfies the loop invariant $Eq(s,t) =$ true iff $s =_E t$. So the algorithm is correct. To estimate the running time $T_{TestEq}(n)$, where $n = |s| + |t|$, we notice that $TestEq(s', t')$ is called at most once for each $(s', t') \in \mathrm{Sub}(s) \times \mathrm{Sub}(t)$. So there are at most n^2 calls of *TestEq* on input(s,t). Each call of *TestEq* costs constant time since the arity of all $f \in F$ and the size of all G_f are bounded by a constant independent of n. Hence algorithm *Eq-Test* needs time $O(n^2)$. $\square$

We now comment on some implementation details to enhance efficiency. The basic idea is to propagate more knowledge from recursive calls of *TestEq*. The first idea is to insert the following test before the line "**for** all $\pi \in G_f \ldots$":

If for some i and all j we have $\mathrm{Treated}(s_i, t_j) \wedge \neg Eq(s_i, t_j)$ then $s \neq_E t$.

The second idea is to exploit the fact that $=_E$ is an equivalence relation, i.e. to use its symmetry and transitivity. So we propose to maintain the current part of the E-equality by the well known *Union-Find* algorithm [CLR96]. Then in algorithm *TestEq* the command $Eq(s', t') :=$ true is to be replaced by $Union(s', t')$. For later use we have $s' =_E t'$ iff $Find(s') = Find(t')$. So the vector Eq is not used any more.

In the following we will frequently have to test whether $t' =_E t''$ holds for $t', t'' \in \mathrm{Sub}(s) \cup \mathrm{Sub}(t)$. So it is recommended to preprocess $*(s,t)$ so that for all $t', t'' \in \mathrm{Sub}(*(s,t))$ we have $t' =_E t''$ iff $\mathit{Find}(t') = \mathit{Find}(t'')$. This preprocessing can be done in time $O(n^2)$ by an algorithm similar to *TestEq*: We have to test whether $t' =_E t''$ bottom up in $*(s,t)$ for all t', t'' such that $\mathrm{top}(t') = \mathrm{top}(t'')$ and $|t'| = |t''|$ and $\mathrm{depth}(t') = \mathrm{depth}(t'')$. If we finally put $Eq(t', t'') = \mathrm{true}$ iff $\mathit{Find}(t') = \mathit{Find}(t'')$ then we can test E-equality on $\mathrm{Sub}(*(s,t))$ in one step.

Next we consider matching modulo E. From Lemma 3.1 we immediately get

Lemma 3.2

a) $\sigma(f(s_1, \ldots, s_k)) = f(t_1, \ldots, t_k)$ iff
$\sigma(s_{\pi(i)}) =_E t_i$ for some $\pi \in G_f$ and all $i = 1, \ldots, k$

b) If $\sigma(s) =_E t$ and $s \notin V$ then $\mathrm{top}(s) = \mathrm{top}(t)$ and $|s| \leq |t|$ and $\mathrm{depth}(s) \leq \mathrm{depth}(t)$. $\Box$

Here we give an algorithm to decide for given s and t whether there is a σ such that $\sigma(s) =_E t$ and, in the positive case, compute such a σ. This is sufficient for doing rewriting and deduction modulo E. If one is interested in computing a complete set of E-matchers from s to t one can apply the techniques developed later for unification modulo E.

Note that already the matching problem modulo commutativity is NP-complete [BKN85], so we do not expect to find a polynomial time algorithm for matching modulo a set E of permutation equations. It is easy to show that any complete set of E-matchers from s to t may grow exponentially in $n = |s| + |t|$ (see below). So it is in general by far less expensive to compute just one E-matcher than a complete set of E-matchers.

To find a "solution" σ such that $\sigma(s) =_E t$ we build a search tree with nodes labeled by a constraint C such that for any C we have: C is solvable iff one of its sons is solvable. We are going to make this precise next.

Definition 3.1
A *constraint* C is a set of equations. It is *E-solvable* if there is a σ such that $\sigma(s) =_E \sigma(t)$ for all $s = t$ in C. Then σ is an *E-solution* of C and $\mathrm{Sol}(C)$ is the set of all E-solutions of C. C is in *solved form* if it is of the form $C = \{x_i = t_i \mid i = 1, \ldots, m\}$ such that for all $i = 1, \ldots m$ we have (1) $x_i \not\equiv x_j$ for $i \neq j$ and (2) $x_i \notin \mathrm{Var}(t_j)$ for $i \leq j \leq m$. C is a *match constraint* if C contains only equations $s = t$ such that t is ground.

If C is in solved form then it defines a substitution $\sigma = \sigma_C$ as usual. If t is ground then $\sigma(t) \equiv t$ and s is E-matchable to t iff $C = \{s = t\}$ is E-solvable. Let

$\sigma(C) = \{\sigma(s) = \sigma(t) \mid s = t \text{ in } C\}$. For pruning a match constraint we use the following lemma.

Lemma 3.3
Let C be a match constraint.

a) $C \cup \{f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k)\}$ is E-solvable
iff $C \cup \{s_{\pi(i)} = t_i \mid i = 1, \ldots, k\}$ is E-solvable for some $\pi \in G_f$.

b) Assume $s \notin V$. $C \cup \{s = t\}$ is not E-solvable if $\mathrm{top}(s) \neq \mathrm{top}(t)$ or $|s| > |t|$ or $\mathrm{depth}(s) > \mathrm{depth}(t)$.

c) Let $\sigma = \{x \leftarrow t\}$. Then $C \cup \{x = t\}$ is E-solvable iff $\sigma(C) \cup \{x = t\}$ is E-solvable.

d) Let $s =_E t$. Then $C \cup \{s = t\}$ is E-solvable iff C is E-solvable.

e) Let t, t' be ground terms and assume $s = t$ is in C.
If $t =_E t'$ then $C \cup \{s = t'\}$ is E-solvable iff C is E-solvable.
If $t \neq_E t'$ then $C \cup \{s = t'\}$ is not E-solvable.

Proof: a) and b) follow from Lemma 3.2 and c) to e) are easy to see. $\square$

Next we give a non-deterministic algorithm for testing whether or not there is a σ such that $\sigma(s) =_E t$. It can easily be expanded to a deterministic one that builds up the whole search space for finding an E-solution of $C = \{s = t\}$. We assume without loss of generality that t is ground. Then only match constraints will appear in the next algorithm.

Algorithm 3.2
Match-Test
Input: $s, t \in \mathrm{Term}(F, V)$ where t is ground.
Output: YES and an E-match if $C = \{s = t\}$ is E-solvable.
NO if $C = \{s = t\}$ is not E-solvable.

Match-Test$(s,t) =$
$C := \{s = t\}$
loop := true
while *loop* **do**
  MPrune(C)
  if $C = \perp$ **then** $\{$*loop* := false ; OUTPUT: NO$\}$ **fi**
  if C is in solved form **then** $\{$*loop* := false ; OUTPUT: YES, $\sigma_C\}$ **fi**
  choose $s \equiv f(s_1, \ldots, s_k) = t \equiv f(t_1, \ldots, t_k)$ in C and $\pi \in G_f$
  $C := (C - \{s = t\}) \cup \{s_{\pi(i)} = t_i \mid i = 1, \ldots, k\}$
od

MPrune

MPrune(C) =
while $s = t$ in C with $s =_E t$ **do** $C := C - \{s = t\}$ **od**
while $s = t, s = t'$ in C with $t =_E t'$ **do** $C := C - \{s = t'\}$ **od**
if $s = t$ in C with $\{\text{top}(s) \neq \text{top}(t)$ and $s \notin V\ \}$ or $|s| > |t|$ or $\text{depth}(s) > \text{depth}(t)$
  or $s = t, s = t'$ in C with $t \neq_E t'$
  or $s = t$ in C with s ground and $s \neq_E t$
 then $C := \perp$ **fi**

Theorem 3.2
Let E be a fixed set of permutation equations and T its table as in section 2. Algorithm *Match-Test*=*Match-Test*(E, T, s, t) is correct. The matchability problem modulo E may be NP-complete.

Proof: Algorithm *Match-Test* is correct by Lemma 3.3. Since it is very similar to syntactic matching, it is easy to see that it needs polynomial time. So matchability modulo E is in NP. Since already matchability modulo commutativity is NP-hard [BKN85], the problem is NP-complete. □

Remark: The construction of [BKN85] uses a function symbol h of arbitrarily large arity m. But h can be chosen to be binary. Then take $s \equiv h(s_1, h(s_2, \ldots, h(s_{m-1}, s_m) \ldots))$ and $t \equiv h(t_1, h(t_2, \ldots, h(t_{m-1}, t_m) \ldots))$ in [BKN85].

We now come to unification modulo E. Here for many applications one needs not only to test whether s and t are unifiable modulo E but to compute a complete set of unifiers modulo E. We repeat some well known definitions:

Let C be a set of equations and Σ a set of substitutions. We write $\sigma =_E \sigma'$ if $\sigma(x) =_E \sigma'(x)$ for all $x \in V$. Σ is a *complete set of unifiers modulo E* for C if (1) each $\sigma \in \Sigma$ is an E-solution of C and (2) for each E-solution σ of C there is a $\sigma' \in \Sigma$ and a τ such that $\sigma =_E \tau \circ \sigma'$. Then Σ is a $\text{CSU}_E(C)$. If $C = \{s = t\}$ then Σ is a complete set of unifiers modulo E for s and t.

For computing a $\text{CSU}_E(\{s = t\})$ we use the following Lemma.

Lemma 3.4

a) $\sigma(f(s_1, \ldots, s_n)) =_E \sigma(f(t_1, \ldots, t_k)$ iff
 $\sigma(s_{\pi(i)}) =_E \sigma(t_i)$ for some $\pi \in G_f$ and all $i = 1, \ldots, k$.

b) Assume $s, t \notin V$. If $\sigma(s) =_E \sigma(t)$ then $\text{top}(s) = \text{top}(t)$.

c) Assume $s \equiv x$. If $x \in \text{Var}(t)$ and $t \not\equiv x$ then there is no E-unifier for s and t. If $x \notin \text{Var}(t)$ and $\sigma = \{x \leftarrow t\}$ then $\{\sigma\}$ is a $\text{CSU}_E(\{x = t\})$.

d) If $s =_E t$ then $\{id\}$ is a $\text{CSU}_E(\{s = t\})$.

Proof: a) and b) follow directly from Lemma 3.1 and c) and d) are easy to prove. $\square$

This lemma directly leads to a non-deterministic algorithm for deciding unifiability modulo E of s and t. This is analogous to the case of matchability modulo E. For pruning a constraint C we use the following lemma, parts b), c) and e).

Lemma 3.5
Let C be a constraint

a) $C \cup \{f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k)\}$ is E-solvable iff $C \cup \{s_{\pi(i)} = t_i \mid i = 1, \ldots, k\}$ is E-solvable for some $\pi \in G_f$.

b) Assume $s, t \notin V$. $C \cup \{s = t\}$ is not E-solvable if $\mathrm{top}(s) \neq \mathrm{top}(t)$.

c) Assume $s =_E t$. $C \cup \{s = t\}$ is E-solvable iff C is E-solvable.

d) Let $\sigma = \{x \leftarrow t\}$. If $x \in \mathrm{Var}(t)$ and $t \not\equiv x$ then $C \cup \{x = t\}$ is not E-solvable. If $x \notin \mathrm{Var}(t)$ then $C \cup \{x = t\ \}$ is E-solvable iff $\sigma(C) \cup \{x = t\}$ is E-solvable.

e) $C \cup \{x = t_1, x = t_2\}$ is E-solvable iff $C \cup \{x = t_1, t_1 = t_2\}$ is E-solvable.

Proof: This follows from Lemma 3.4. $\square$

If $G_f = \langle id \rangle$ for all $f \in F$ then E-unification reduces to syntactic unification, i.e., to testing whether $\sigma(s) \equiv \sigma(t)$ for some σ. The standard algorithm for syntactic unification is to transform C into C' in solved form by applying the deduction steps indicated by Lemma 3.5 a) to d). But that may need exponential time. There are well known techniques to design polynomial time algorithms for syntactic unification (see e.g. [BN98]). One idea is to replace deduction step d) of Lemma 3.5 by deduction step e) and testing C' for cycle-freeness. We adopt this technique.

For computing a $\mathrm{CSU}_E(s = t)$, starting with $C = \{s = t\}$ the deduction steps a)-c) and e) transform C into a set $\mathrm{Con}(s, t)$ of constraints each of the form $C' = \{x_i = t_i \mid i = 1, \ldots, m\}$ with $x_i \not\equiv x_j$ for $i \neq j$. We call such a C' *pre-solved*. Notice that during this transformation only equations $t_1 = t_2$ appear such that $t_1, t_2 \in \mathrm{Sub}(s) \cup \mathrm{Sub}(t)$. So, if we preprocess (s, t) as indicated after Theorem 3.1 we can test E-equality of these t_1, t_2 in one step. This preprocessing will also help us to memorize those $\mathrm{Con}(t_1, t_2)$ that are already computed.

It is advisable to eliminate unsolvable constraints early in the transformation from $C = \{s = t\}$ into $\mathrm{Con}(s, t)$. Unsolvability of a constraint can be detected by the test on cycle-freeness. We are going to describe this.

We reduce a constraint C to $\mathrm{red}(C)$ by first performing **while** $C = C' \cup \{x = y\}$ with $x \not\equiv y, x \in \mathrm{Var}(C')$ **do** $\sigma := \{x \leftarrow y\}, C := \sigma(C') \cup \{x = y\}$ **od** and then removing all equations of the form $x = x$. We define $\mathrm{graph}(C) = (V_C, E_C)$ by $V_C = \mathrm{Var}(C)$ and $E_C = \{(x, y) \mid x = t \text{ in } \mathrm{red}(C), y \in \mathrm{Var}(t), y \not\equiv t\}$. We call C

cycle-free if graph(C) has no cycle. Testing for cycle-freeness can be done in linear time (see [CLR96]).

Lemma 3.6

a) If C is not cycle-free then C is E-unsolvable.

b) If C is pre-solved and cycle-free then $\mathrm{Sol}(C) = \mathrm{Sol}(\mathrm{red}(C))$ and $\mathrm{red}(C)$ is in solved form. So C defines the E-solution $\sigma = \sigma_{\mathrm{red}(C)}$ of C.

Proof: We have clearly $\mathrm{Sol}(C) = \mathrm{Sol}(\mathrm{red}(C))$.

a) If graph(C) has a cycle leading from $x = t'$ to $y = t''[x]_p$ in $\mathrm{red}(C)$ then one can easily construct a term t with $x \in \mathrm{Var}(t), x \not\equiv t$ and $\sigma(x) =_E \sigma(t)$ for each $\sigma \in \mathrm{Sol}(C)$. So $\mathrm{Sol}(C) = \emptyset$.

b) Define $\succ_0$ on V_C by $y \succ_0 x$ iff there is a path from x to y in graph(C). Since graph(C) has no cycle, $\succ_0$ is a partial order. Hence $\succ_0$ can be embedded into a linear order $\succ$ on V_C. If $x = t, y = t'$ in $\mathrm{red}(C)$ and $y \succ x$ then $x \notin \mathrm{Var}(t')$. For, $x \in \mathrm{Var}(t')$ implies $x \succ_0 y$. So $\mathrm{red}(C)$ is in solved form. $\square$

Now a non-deterministic algorithm *Unif-Test* can be given. It transforms $C = \{s = t\}$ into a pre-solved C' by guessing the π in deduction step a) of Lemma 3.5.

Algorithm 3.3
Unif-Test
Input: $s, t \in \mathrm{Term}(F, V)$.
Question: Is there a σ such that $\sigma(s) =_E \sigma(t)$?

Unif-Test$(s,t) =$
$C := \{s = t\}$; *loop* := true
while *loop* **do**
  UPrune(C)
  if $C = \perp$ **then** $\{$*loop* := false ; OUTPUT: NO$\}$ **fi**
  if C is in solved form **then** $\{$*loop* := $false$; OUTPUT: YES, $\sigma = \sigma_C\}$ **fi**
  choose $s \equiv f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k) \equiv t$ in C and $\pi \in G_f$
  $C := C - \{s = t\} \cup \{s_{\pi(i)} = t_i \mid i = 1, \ldots, k\}$
od

UPrune
UPrune$(C) =$
while $t = x$ in C with $t \notin V$ **do** $C := (C - \{t = x\}) \cup \{x = t\}$ **od**
while $C = C' \cup \{x = t_1, x = t_2\}$ **do** $C := C' \cup \{x = t_1, t_1 = t_2\}$ **od**
if ($s = t$ in C with $s, t \notin V$ and $\mathrm{top}(s) \neq \mathrm{top}(t)$) or ($C$ is not cycle-free)
  then $C := \perp$
  else **while** $s = t$ in C with $s =_E t$ **do** $C := C - \{s = t\}$ **od** **fi**

The result of algorithm *UPrune* is described by

Lemma 3.7
Assume C' is the constraint produced by *UPrune*(C). If $C' = \bot$ then C is not E-solvable. If $C' \neq \bot$, then we have $\mathrm{Sol}(C) = \mathrm{Sol}(C')$. Each equation in C is of the form $x = t$ or $s = t$ with $\mathrm{top}(s) = \mathrm{top}(t) \in F$. $\square$

Theorem 3.3
Let E be a fixed set of permutation equations and T its table as in section 2. Algorithm *Unif-Test*=*Unif-Test*(E,T,s,t) is correct. The unification problem modulo E may be NP-complete.

Proof: *Unif-Test* is correct by Lemma 3.5 and Lemma 3.7. Since the (nondeterministic) algorithm is almost identical to syntactic unification, it needs polynomial time. (Notice that in the transformation of C into pre-solved form only equations $t_1 = t_2$ with $t_1, t_2 \in \mathrm{Sub}(s) \cup \mathrm{Sub}(t)$ appear. The number of these equations is bounded by n^2 with $n = |s| + |t|$. Algorithm *UPrune*(C) needs polynomial time.) So unification modulo E is in NP. Since already matchability modulo commutativity is NP-hard [BKN85], the problem is NP-complete. $\square$

One can try to compute a $\mathrm{CSU}_E(s = t)$ by transforming *Unif-Test* into a deterministic algorithm that builds up the whole search tree and collects all solutions. In principle this seems to be unavoidable. We demonstrate this by an example.

Example 3.2
Let s and t be described by the full binary tree of depth m with all inner nodes labeled by f. The $n = 2^m$ leaves of s are labeled from left to right by $x_1, \ldots, x_n$ and those of t are replaced by $g^1(a), \ldots, g^n(a)$. Assume $E = \{f(x, y) = f(y, x)\}$. Then E-solving $C = \{s = t\}$ is a matching problem modulo E. When building up the full search tree for finding all E-solutions of C each branch in the tree leads to a solution $\sigma_\varphi = \{x_i \leftarrow g^{\varphi(i)}(a) \mid i = 1, \ldots, n\}$, where $\varphi \in \mathrm{Sym}(n)$. Let Φ be the set of these φ and $[s]_E = \{s' \mid s =_E s'\}$. There is a bijection $\alpha : \Phi \to [s]_E$ such that $\sigma_\varphi(s_\varphi) \equiv t$ where $s_\varphi = \alpha(\varphi) \in [s]_E$. Induction on m shows that $[s]_E$ contains 2^n elements. So there are 2^n E-matchers from s to t. Notice that $|s| + |t| = O(n^2)$.

As a consequence, the number of E-matchers from s to t may grow exponentially in $|s| + |t|$. Hence also each $\mathrm{CSU}_E(s = t)$ may grow exponentially in $|s| + |t|$. $\square$

Let us now comment on the design decisions for our algorithm *Unify*(s, t) to compute a $\mathrm{CSU}_E(s, t)$. When transforming *Unif-Test* naively into a deterministic algorithm then subproblems $C' = \{t' = t''\}$ are solved repeatedly for some fixed $t', t'' \in \mathrm{Sub}(*(s, t))$, though in different contexts. To avoid this we follow the strategy of algorithm *Eq-Test* and store the solution of C' in *Con*(t', t'') as soon as it is computed. To do so we first create the necessary data structure (graph) to support this storing and to support testing for E-equality as mentioned above.

For describing *Unify* we use the following notations and subalgorithms: C always denotes a constraint and Con denotes a set of constraints. $UCompat$ is the compatibility predicate

$UCompat(s, t, \pi)$ iff

For all $i = 1, \ldots, ar(f)$: $s_{\pi(i)} \in V$ or $t_i \in V$ or $\mathrm{top}(s_{\pi(i)}) = \mathrm{top}(t_i)$.

$Preprocess(s, t)$ does the preprocessing of (s, t) and initializes $\mathrm{Treated}(t', t'') :=$ false for all $t', t'' \in \mathrm{Sub}(*(s, t))$.

$Combine(C, Con)$ is a function that produces $Con' = \{C_0 \mid C_0 = UPrune(C \cup C'), C' \in Con, C_0 \neq \perp\}$. For *UPrune* see Algorithm 3.3.

$PreUnify(Con)$ transforms each $C \in Con$ into a set of pre-solved constraints.

$Tran(s, t)$ delivers in $Con(s, t)$ a set of constraints simpler than $C = \{s = t\}$ but in general not pre-solved.

Before presenting the algorithms in detail we shortly describe in words how they work. To describe *Trans(s,t)* assume $s \equiv f(s_1, ..., s_k)$ and $t \equiv f(t_1, ...t_k)$. (The other cases of the structure of s and t are easy.) Then *Con(s,t)* is the union of all $Con\pi$ with $\pi \in G_f$. Here $Con\pi$ is the combination of all $C_j \in Con(s_{\pi(j)}, t_j), j = 1, ..., k$. One could compute all $Con(s_{\pi(j)}, t_j)$ first and then put $Con* = \{C \mid C = UPrune(C_1 \cup ... \cup C_k), C_j \in Con(s_{\pi(j)}, t_j)\}$. Then we have $Con\pi = Con*$ but $Con*$ may contain some $C = \perp$. For reasons of efficiency we compute instead $Coni = \{C \mid C = UPrune(C_1 \cup ... \cup C_i), C_j \in Con(s_{\pi(j)}, t_j), C \neq \perp\}$ for $i = 1, ..., k$. This will eliminate early all $C \in Con*$ with $C = \perp$. Specifically, if $Coni$ becomes empty then immediately $Con\pi$ becomes empty. Finally, for $i = k$ we have $Coni = Con\pi$.

If, in general, $Sol(Con)$ denotes the union of all $Sol(C)$ with $C \in Con$ then *Trans(s,t)* satisfies the loop invariant that $Sol(\{s = t\}) = Sol(Con(s, t))$. Some $C \in Con(s, t)$ again may contain equations $s = t$ with $top(s) = top(t) \in F$. These equations are processed by *PreUnify(Con)*. In the loop of this algorithm each $C = C' \cup \{s = t\}$ in Con with $top(s) = top(t) \in F$ is transformed into $Con_C = Combine(C', Con(s, t))$. Since $Sol(C) = Sol(Con_C)$ we have the loop invariant for *PreUnify* that $Sol(Con)$ is fixed.

Algorithm 3.4
Unify

Input: $s, t \in \mathrm{Term}(F, V)$
Output: Con, describing a $\mathrm{CSU}_E(s, t)$

$Unify(s, t) =$
$Preprocess(s, t)$
$C := \{s = t\}$; $UPrune(C)$;
if $C = \perp$ **then** $Con := \varnothing$ **else** $Con := \{C\}$; $PreUnify(Con)$ **fi**
OUTPUT : Con

UPrune

UPrune(C) =
while $t = x$ in C with $t \notin V$ **do** $C := (C - \{t = x\}) \cup \{x = t\}$ **od**
while $C = C' \cup \{x = t_1, x = t_2\}$ **do** $C := C' \cup \{x = t_1, t_1 = t_2\}$ **od**
if ($s = t$ in C with $s, t \notin V$ and $\mathrm{top}(s) \neq \mathrm{top}(t)$) or ($C$ is not cycle-free)
  then $C := \bot$
  else **while** $s = t$ in C with $s =_E t$ **do** $C := C - \{s = t\}$ **od** **fi**

PreUnify

PreUnify(Con) =
while $\exists C \in Con, s = t$ in $C, \mathrm{top}(s) = \mathrm{top}(t) \in F$ **do**
  $Con := Con - \{C\}$
  $C := C - \{s = t\}$
  if not Treated(s, t) **then** *Trans*(s, t) **fi**
  $Con := Con \cup Combine(C, Con(s, t))$
od

Trans

Trans(s, t) =
$Con := \varnothing$
if $s \in V$ or $t \in V$ **then** $Con := \{UPrune(\{s = t\})\}$ **fi**
if $\mathrm{top}(s) = \mathrm{top}(t) \in F$ **then**
  let $s \equiv f(s_1, \ldots, s_k), t \equiv f(t_1, \ldots, t_k)$
  for all $\pi \in G_f$ such that $UCompat(s, t, \pi)$ **do**
    $Con\pi := \{\varnothing\};\ i := 1$
    while $i \leq k$ and $Con\pi \neq \varnothing$ **do**
      $Coni := \varnothing$
      if not Treated$(s_{\pi(i)}, t_i)$ **then** *Trans*$(s_{\pi(i)}, t_i)$ **fi**
      for all $C \in Con(s_{\pi(i)}, t_i)$ **do** $Coni := Coni \cup Combine(C, Con\pi)$ **od**
      $Con\pi := Coni;\ i := i + 1$
    od
    $Con := Con \cup Con\pi$
  od **fi**
$Con(s, t) := Con$; Treated(s, t) := true

We assume here that E is fixed and T is already computed. Again, the enumeration of all $\pi \in G_f$ such that $UCompat(s, t, \pi)$ is supported by algorithm $\mathcal{A}$ from section 2 and that this also takes into account for which i the π-loop was interrupted. See Example 3.1 to demonstrate that.

Theorem 3.4
Algorithm *Unify*(s, t) stops and produces a $\mathrm{CSU}_E(s, t)$. □

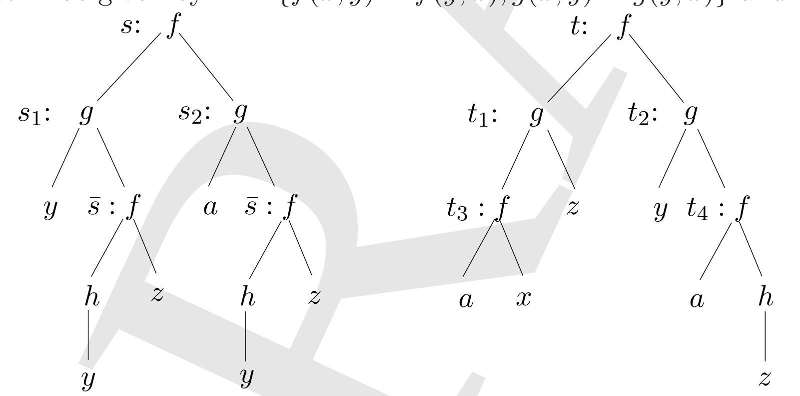
We want to justify that algorithm *Unify* is correct and terminates. Let $Sol(Con)$ be the union of all $Sol(C)$ with $C \in Con$. Then $\mathrm{Sol}(\{s = t\}) = \mathrm{Sol}(Con(s,t))$. The while-loop in *PreUnify* satisfies the loop invariant that $Sol(Con)$ is fixed. All $C \in Con$ in the output of *PreUnify* are pre-solved and cycle-free. So the output of $\mathit{Unify}(s,t)$ describes a $\mathrm{CSU}_E(s,t)$. To prove the termination of *Unify*, we show that the while-loop in *PreUnify* decreases Con in a well-founded order: For a constraint C let $\mathrm{set}(C) = \{s,t \mid s = t \text{ in } C\}$ be the set of all terms in C and let $mult(C)$ be the corresponding multiset. For any ordered set $(M, >)$ we denote by $(\mathcal{M}(M), >_{mult})$ the multiset extension of $(M, >)$, see [BN98]. Let $>_{st}$ be the subterm order. We define the well-founded order $\succ$ on constraints by

$C \succ C'$ iff $mult(C) >_{st,mult} mult(C')$.

This defines the well-founded order $\succ_{mult}$ on sets Con of constraints.
Now let C' be the result of $\mathit{UPrune}(C)$. We have $\mathrm{set}(C') \subseteq \mathrm{set}(C)$ and so $\{s = t\} \succ C$ implies $\{s = t\} \succ C'$. This gives $\{s = t\} \succ C_0$ for all $C_0 \in Con(s,t)$ produced by $\mathit{Trans}(s,t)$ and hence $\{\{s = t\}\} \succ_{mult} Con(s,t)$. Now it is easy to see that the while-loop in algorithm *PreUnify* decreases Con in the order $\succ_{mult}$. This proves that algorithm *Unify* terminates.

One can check that $\mathit{Unify}(s,t)$ runs in polynomial time provided that the number of constraints C generated is bounded by a polynomial in $n = |s| + |t|$. (Notice that *Trans* is called at most n^2 times.)

Example 3.3
Let E be given by $E = \{f(x,y) = f(y,x), g(x,y) = g(y,x)\}$ and



The algorithm $\mathit{Trans}(s,t)$ basically computes the sets of constraints $Con(s',t')$ left-to-right bottom-up. This gives (constraints already in solved form)
$Con(\bar{s},t_3) : \{\{h(y) = a, z = x\}, \{h(y) = x, z = a\}\} \rightsquigarrow \{\{z = a, x = h(y)\}\}$
$Con(\bar{s},t_4) : \{\{h(y) = a, z = h(z)\}, \{h(y) = h(z), z = a\}\} \rightsquigarrow \{\{y = a, z = a\}\}$
$Con(s_1,t_1) : \{\{z = a, y = a, x = h(a)\}\}$
$Con(s_1,t_2) : \{\{z = a, y = a\}\}$
$Con(s_2,t_1) : \{\{z = a, x = h(y)\}\}$

$Con(s_2, t_2) : \{\{z = a, y = a\}\}$
$Con(s, t) : \{\{z = a, y = a, x = h(a)\}\}$
Unify(s, t) produces only one constraint, it is solved. So there is exactly one E-unifier of s and t: $\sigma = \{x \leftarrow h(a), y \leftarrow a, z \leftarrow a\}$.

Now let $s_0 \equiv f(x', x')$ and $t_0 \equiv f(s, t)$. We call *Unify*(s_0, t_0). Then *PreUnify*$(\{\{s_0 = t_0\}\})$ calls *Trans*(s_0, t_0) and *Trans*(s_0, t_0) produces $Con(s_0, t_0) = \{C\}$ with $C = \{x' = t, s = t\}$. Then *PreUnify* calls *Trans*(s, t) which produces $Con(s, t) = \{C'\}$ with $C' = \{x = h(a), y = a, z = a\}$. Combining $\{x' = t\}$ with $Con(s, t)$ leads to $Con = \{C''\}$ with $C'' = \{x = h(a), y = a, z = a, x' = t\}$ which is in solved form. Now *PreUnify* stops with output $Con = \{C''\}$. So there is exactly one (up to E-equality) E-unifier $\sigma_0 = \sigma_{C''}$ for s_0 and t_0. $\square$

Let us add a comment on the implementation of algorithm *Unify*. At the heart of this algorithm are the subalgorithms *PreUnify* and *Trans*. Algorithm *Trans*(s, t) transforms $C = \{s = t\}$ into a set $Con(s, t)$ of constraints simpler than C but in general not in solved form. The transformation of $Con(s, t)$ into a set of constraints in solved form is done by *PreUnify*. If $Con(s, t)$ is reused, this transformation into solved form has to be repeated. This can be avoided by inserting the command *PreUnify*(Con) before the last line in algorithm *Trans*. This modification makes the algorithms *Trans* and *PreUnify* mutually recursive, but does not destroy the correctness of *Unify*. We believe that it increases the efficiency of the unification algorithm.

Up to now we have considered the problems of E-equality, E-matchability and E-unification only for a fixed set E of permutation equations. We now come to the generalized version of these problems where E is allowed to be part the input.

Theorem 3.5
The E-unifiability for permutation equations E is NP-complete.

Proof: The proof of Theorem 3.4 carries over to this problem: Let $n = |E| + |s| + |t|$ be the length of the input for *Unif-Test*. The number of decompose steps "choose s, t and $\pi \in G_f \dots$" is bounded by a polynomial in n. For any of these steps the choice of s and t is don't-care non-deterministic. Choose an arbitrary $\pi \in \mathrm{Sym}(k)$ and test whether $\pi \in G_f$. This can be done in polynomial time by Theorem 2.1. Algorithm *UPrune* runs in deterministic polynomial time. So algorithm *Unif-Test* runs in non-deterministic polynomial time. $\square$

Theorem 3.6
The E-equality for permutation equations E is in NP, but it is at least as hard as the graph isomorphism problem. The second claim already holds for those E that contain two equations.

Proof: The first claim follows from Theorem 3.5. To prove the second claim we use the results of section 2, especially Lemma 2.2: Consider two graphs $\mathcal{G}_i = (\mathbb{N}_k, \bar{E}_i), i = 1, 2$. Let $\pi_i \in \mathrm{Sym}(k)$ be given by $\pi_1(1) = 2, \pi_1(2) = 1$ and $\pi_1(j) = j$ for $j = 3, \ldots, k$ and $\pi_2(j) = j+1$ for $j < k$ and $\pi_2(k) = 1$. Then $\langle \pi_1, \pi_2 \rangle = \mathrm{Sym}(k)$. Let $n = k^2$. Let $\_^* : \mathrm{Sym}(k) \to \mathrm{Sym}(n)$ be given as in section 2. Let $\mathrm{sig} = (F, \alpha)$ with $F = \{f, 0, 1\}$, where $\alpha(f) = n$, $\alpha(0) = \alpha(1) = 1$ and let $E = \{f(x) = f(\pi_1^*(x)), f(x) = f(\pi_2^*(x))\}$, where $x = (x_1, \ldots, x_n)$.
Then $G_f = \langle \pi_1^*, \pi_2^* \rangle = \{\pi^* \mid \pi \in \mathrm{Sym}(k)\}$. Let $u_i \in \{0,1\}^n$ be the characteristic word (also considered as a 0-1-vector of length n) of $\mathcal{G}_i, i = 1, 2$. Let $s \equiv f(u_1)$ and $t \equiv f(u_2)$ then we have
$s =_E t$ iff $\pi^*(u_1) \equiv u_2$ for some $\pi^* \in G_f$
iff $\pi^*(u_1) \equiv u_2$ for some $\pi \in \mathrm{Sym}(k)$
iff $\mathcal{G}_1$ and $\mathcal{G}_2$ are isomorphic.
So the graph isomorphism problem is polynomial time reducible to the E-equality problem. $\square$

Notice that the set of axioms $E = E(\mathcal{G}_1, \mathcal{G}_2)$ depends on $\mathcal{G}_1$ and $\mathcal{G}_2$ in that $ar(f) = k^2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ have k vertices. So $E(\mathcal{G}_1, \mathcal{G}_2)$ becomes larger in some sense if the $\mathcal{G}_i$ grow in size. Hence, Theorem 3.6 does not contradict Theorem 3.1.

In our setting, to any function symbol f only one permutation group G_f is associated. This assumption is reasonable for computing modulo E. In [dlTE03] the NP-completeness of several problems for permutative equations (see below) is shown. Translated to our context, these results imply that the E-equality problem is NP-complete if several permutation groups can be associated to some function symbols. We did not try hard to transform the proof of [dlTE03] to our more special setting.

4 Permutative axioms

The ideas presented in Section 3 can easily be translated (in a restricted form) to the situation where E is a set of permutative equations. How to compute in equational theories defined by permutative equations was introduced in [AP01].

We start with some notations. A *context* $c \equiv c[\Box_1, \ldots, \Box_k]$ is a term in $\mathrm{Term}(F, V \cup \{\Box_1, \ldots, \Box_k\})$ such that $\mathrm{top}(c) \in F$ and each $\Box_i$ appears exactly once. For $t_1, \ldots, t_k \in \mathrm{Term}(F, V)$ $c[t_1, \ldots, t_k]$ results from $c[\Box_1, \ldots, \Box_k]$ by replacing each $\Box_i$ by t_i. We write $c \equiv c_f$ to indicate that $\mathrm{top}(c_f) = f$.

A *permutative equation* is of the form $c[\bar{x}] = c[\pi(\bar{x})]$ with $\pi \in \mathrm{Sym}(k)$. As an example, $f(g(x), h(y, z)) = f(g(z), h(y, x))$ is a permutative equation with context $c \equiv f(g(\Box_1), h(\Box_2, \Box_3))$ and $\pi(1) = 3, \pi(2) = 2$ and $\pi(3) = 1$. We denote by E_c any finite set of such equations: $E_c = \{c[\bar{x}] = c[\pi_i(\bar{x})] \mid i = 1, \dots, m\}$. Let I be a set of contexts and let E be the union of the $E_c,\ c \in I$.

We mention that for any context c we have only one set E_c of permutative equations. This is a mild, but reasonable, restriction compared to [AP01].

Compared to permutation axioms, it is not so easy to compute efficiently with the full equivalence classes $[t]_E$. In [AP01] therefore algorithms are presented to compute with restricted equivalence classes $[t]_{\hat{E}}$ for "stratified terms" t.

To explain this, for each E_{c_f} a new function symbol $\hat{f} = \hat{f}_c$ is introduced. Let $\hat{F}$ be the set of these $\hat{f}$. Now E_{c_f} is replaced by $E_{\hat{f}} = \{\hat{c}_f[\bar{x}] = \hat{c}_f[\pi_i(\bar{x})] \mid i = 1, \dots, m\}$ where $\hat{c}_f$ results from c_f by replacing the top symbol f with $\hat{f}_c$. Let $\hat{E}$ by the union of these $E_{\hat{f}}$. (Notice that in $\hat{c}_f$ only the top symbol $\hat{f}$ is in $\hat{F}$.) Let $G_{\hat{f}_c} = \langle \pi \mid \hat{c}_f[\bar{x}] = \hat{c}_f[\pi(\bar{x})] \text{ in } E_{\hat{f}} \rangle$, where $\bar{x} = (x_1, \dots, x_k)$.

A term $t \in \mathrm{Term}(F \cup \hat{F}, V)$ is *stratified* if for each $s \equiv \hat{f}(s_1, \dots, s_n) \in \mathrm{Sub}(t)$ there is a match from $\hat{c}_f[\bar{x}]$ to s, i.e. s is of the form $s \equiv \hat{c}_f[s'_1, \dots, s'_k]$. Notice that for no two distinct $c, c' \in I$ the terms $\hat{c}[x_1, \dots, x_k]$ and $\hat{c}'[x_1, \dots, x_k]$ are unifiable, since they have different top symbols in $\hat{F}$. So for any stratified term s there is at most one $c \in I$ such that s is of the form $s \equiv \hat{c}_f[s'_1, \dots, s'_k]$ and applying an equation in $E_{\hat{f}}$ to s leads to $s' \equiv \hat{c}_f[s'_{\pi(1)}, \dots, s'_{\pi(k)}]$ with the same head context $\hat{c}_f$ as s. Hence the applicability of equations in $\hat{E}$ on stratified terms commutes: if $\hat{E} = E_{\hat{f}_1} \cup \dots \cup E_{\hat{f}_n}$ then $s =_{\hat{E}} t$ iff there are terms $t_0, t_1, \dots, t_n$ with $s \equiv t_0, t \equiv t_n$ and $t_{i-1} =_{E_{\hat{f}_i}} t_i$ for $i = 1, \dots, n$. We are interested in algorithms to compute with the equivalence classes $[t] = [t]_{\hat{E}}$ for stratified terms t.

Notation: Let $t \in \mathrm{Term}(F \cup \hat{F}, V)$ be stratified. $um(t)$ results from t by "unmarking" t, i.e. by replacing each $\hat{f}$ with f. Let $Um(t) = \{um(t') \mid t' \in [t]_{\hat{E}}\}$.

Let us shortly comment on the underlying idea of the construction: We have marked a term $t \in \mathrm{Term}(F, V)$ by changing some subterms $t' \equiv f(t_1, \dots, t_k) \in \mathrm{Sub}(t)$ to $\hat{t}' \equiv \hat{f}(t_1, \dots, t_k)$. This marking is intended to restrict the applicability of equations to these t'. This is done by switching from E to $\hat{E}$. An equation in $\hat{E}$ can be applied to a term t at position p only if $\mathrm{top}(t/p) = \hat{f}_c \in \hat{F}$ and $t/p \equiv \hat{c}_f[\tilde{t_1}, \dots, \tilde{t_l}]$. Now $um(t) \in \mathrm{Term}(F, V)$ is the unmarking of $t \in \mathrm{Term}(F \cup \hat{F}, V)$. This way a stratified term t represents the $\hat{E}$-equivalence class $[t]_{\hat{E}}$ and so the set $Um(t) = \{um(t') \mid t' \in [t]_{\hat{E}}\}$ of unmarked terms.

If there are no overlaps between equations in E, i.e., no c_f unifies with any subterm of any c_g, then the approximation of $=_E$ by $=_{\hat{E}}$ is exact, that is we have $[um(t)]_E = Um(t)$ for any stratified term t. In this case the algorithms from below remain correct if we replace $\hat{E}$ by E.

The problems of main interest are as follows

Definition 4.1 (Equality modulo $\hat{E}$)
Input: $s \in \mathrm{Term}(F, V), t \in \mathrm{Term}(F \cup \hat{F}, V)$ stratified.
Question: Does $s \in Um(t)$ hold?

Definition 4.2 (Matchability modulo $\hat{E}$)
Input: $s \in \mathrm{Term}(F, V), t \in \mathrm{Term}(F \cup \hat{F}, V)$ stratified.
Question: Is there a σ such that $\sigma(s) \in Um(t)$?

Definition 4.3 (Unifiability modulo $\hat{E}$)
Input: $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ stratified.
Question: Is there a σ such that $\sigma(s') \equiv \sigma(t')$ for some $(s', t') \in Um(s) \times Um(t)$?
Compute $\Sigma(s, t) = \{mgu(s', t') \mid (s', t') \in Um(s) \times Um(t)\}$.

Notice that $um([t]_{\hat{E}}) \subseteq [um(t)]_E$ for any stratified term t. But, if E consists of permutation axioms only then $[um(t)]_E = um([t]_{\hat{E}})$. So the results of this section generalize those of section 3.

For the next lemma recall that for any $t' \in \mathrm{Sub}(t)$, where t is stratified and $\mathrm{top}(t') \in \hat{F}$, we have exactly one $c_f \in I$ with $t' \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$.

It turns out that the algorithms of section 3 can be adapted to solve these problems for permutative groups as well. Here the memoization techniques used pay off even more. We work out the details for testing equality modulo $\hat{E}$ and give the basic lemmas for the other two problems.

Lemma 4.1
Let $s \equiv f(s_1, \ldots, s_k) \in \mathrm{Term}(F, V)$ and $t \in \mathrm{Term}(F \cup \hat{F}, V)$ where t is stratified.

a) $s \in Um(t)$ iff
$t \equiv f(t_1, \ldots, t_k)$ and $s_i \in Um(t_i)$ for all $i = 1, \ldots, k$ or
$s \equiv c_f[\tilde{s_1}, \ldots, \tilde{s_l}],\ t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and $\tilde{s_i} \in Um(\tilde{t}_{\pi(i)})$ for some $\pi \in G_{\hat{f}_c}$ and all $i = 1, \ldots, l$.

b) If $s \in Um(t)$ then $\mathrm{top}(s) = \mathrm{top}(um((t))$ and $|s| = |t|$.

Proof: By induction on depth(s). See [AP01]. □

This leads to the following algorithm. It uses the compatibility predicate
$EqCompat(s, t, \pi)$ iff
For all $i = 1, \ldots, ar(f)$: $\mathrm{top}(\tilde{s}_{\pi(i)}) = \mathrm{top}(\tilde{t_i})$, $|\tilde{s}_{\pi(i)}| = |\tilde{t_i}|$ and
$\mathrm{depth}(\tilde{s}_{\pi(i)})$ $=\mathrm{depth}(\tilde{t_i})$.

Algorithm 4.1
Eq-Test
Input: $s \in \mathrm{Term}(F, V), t \in \mathrm{Term}(F \cup \hat{F}, V)$ where t is stratified.
Output: YES if $s \in Um(t)$ and NO else

Eq-Test(s,t) =
for all $(s',t') \in \mathrm{Sub}(s) \times \mathrm{Sub}(t)$ **do** $\mathrm{Treated}(s',t') := \mathrm{false}$ **od**
TestEq(s,t)
OUTPUT *Eq*(s,t)

TestEq
TestEq(s,t) =
Eq$(s,t) := \mathrm{false}$
if $\mathrm{top}(s) = \mathrm{top}(um(t))$ and $|s| = |t|$ **then**
  if $s,t \in V \cup F$ **then** $Eq(s,t) := \mathrm{true}$ **fi**
  if $s \equiv f(s_1,\ldots,s_k)$ and $t \equiv f(t_1,\ldots,t_k)$ **then**
    $ok := \mathrm{true};\ i := 1$
    while $i \le k$ and ok **do**
      if not $\mathrm{Treated}(s_i,t_i)$ **then** $TestEq(s_i,t_i)$ **fi**
      if $Eq(s_i,t_i) = \mathrm{false}$ **then** $ok := \mathrm{false}$ **fi**
    od
    if ok **then** $Eq(s,t) := \mathrm{true}$ **fi**
  fi
  if $s \equiv c_f[\tilde{s_1},\ldots,\tilde{s_l}]$ and $t \equiv \hat{c}_f[\tilde{t_1},\ldots,\tilde{t_l}]$ **then**
    for all $\pi \in G_{\hat{f}}$ such that $EqCompat(s,t,\pi)$ **do**
      $ok\pi := \mathrm{true};\ i := 1$
      while $i \le l$ and $ok\pi$ **do**
        if not $\mathrm{Treated}(\tilde{s}_{\pi(i)},\tilde{t_i})$ **then** $TestEq(\tilde{s}_{\pi(i)},\tilde{t_i})$ **fi**
        if $Eq(\tilde{s}_{\pi(i)},\tilde{t_i}) = \mathrm{false}$ **then** $ok\pi := \mathrm{false}$ **fi**
      od
      if $ok\pi$ **then** $\{Eq(s,t) := \mathrm{true};\ Treated(s,t) := \mathrm{true};\ exit\}$ **fi**
    od
  fi
fi
$\mathrm{Treated}(s,t) := \mathrm{true}$

This algorithm *Eq-Test* works very similar to its corresponding algorithm for permutation groups from section 3. Recall that in the previous lemma there is at most one $c_f \in I$ such that $t \equiv \hat{c}_f[\tilde{t_1},\ldots,\tilde{t_l}]$. So the proofs for correctness and the time requirement directly carry over.

Theorem 4.1
Let E be a fixed set of permutative equations and T its table as in section 2. Algorithm *Eq-Test* = *Eq-Test(E,T,s,t)* is correct. It runs in time $O(n^2)$, where $n = |s| + |t|$. So the problem of testing $\hat{E}$-equality for permutative axioms is solvable in polynomial time. $\square$

If E is allowed to be part of the input, then testing equality modulo $\hat{E}$ is in NP, but at least as hard as testing for graph isomorphism. The proof of Theorem 3.6 carries over. For encoding the graph isomorphism problem into the $\hat{E}$-equality problem we here do not need function symbols of arbitrary arity: Using a binary function symbol f' we can decode $f(x_1, \ldots, x_n)$ by $f'(x_1, f(x_2, \ldots, f(x_{n-1}, x_n) \ldots))$. This encoding transforms the set E of permutation equations into a set E' of permutative equations.

We now come to matchability modulo $\hat{E}$. Again, we can assume that t is a ground term.

Lemma 4.2
Assume $s \equiv f(s_1, \ldots, s_k) \in \mathrm{Term}(F, V)$ and $t \in \mathrm{Term}(F \cup \hat{F})$ and t is stratified.

a) $\sigma(s) \in Um(t)$ iff
$t \equiv f(t_1, \ldots, t_k)$ and $\sigma(s_i) \in Um(t_i)$ for all $i = 1, \ldots, k$ or
$t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$, $\mu = mgu(s, c_f)$, $\mu(s) \equiv c_f[\tilde{s_1}, \ldots, \tilde{s_l}]$, $\tau(\tilde{s_i}) \in Um(\tilde{t}_{\pi(i)})$ and $\sigma(x) =_{\hat{E}} \tau(\mu(x))$ for all $x \in Var(s)$ for some τ, some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

b) If $s \in Um(t)$ and $s \notin V$ then $\mathrm{top}(s) \neq \mathrm{top}(um(t))$ and $|s| \leq |t|$.

Proof: This follows from Lemma 4.1. Notice that in case $t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ we have first to instantiate s minimally to $\mu(s)$ such that $\mu(s)$ is an instance of c_f and then to apply the given test whether $\tau(\mu(s)) \in Um(t)$ for some τ. □

Using the techniques of the corresponding algorithm from section 3 and those from algorithm *Eq-Test* one can design a non-deterministic algorithm *Match-Test* for the E-matchability problem that runs in polynomial time.

Theorem 4.2
The matchability problem modulo $\hat{E}$ for permutative axioms E is NP-complete.

Proof: There is a non-deterministic polynomial time algorithm, so the problem is in NP. Since already the matchability problem modulo commutativity is NP-hard, the problem is NP-complete. □

Next, we consider unification modulo E. We first present a lemma that helps to compute a $\mathrm{CSU}_{\hat{E}}(s, t)$ for stratified terms s and t.

Lemma 4.3
Assume $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ are stratified.

a) If $s \equiv f(s_1, \ldots, s_k)$ then
$\sigma(s) =_{\hat{E}} \sigma(t)$ iff
$t \equiv f(t_1, \ldots, t_k)$ and $\sigma(s_i) =_{\hat{E}} \sigma(t_i)$ for all $i = 1, \ldots, k$.

b) If $s \equiv \hat{c}_f[\tilde{s_1}, \ldots, \tilde{s_l}]$ then
$\sigma(s) =_{\hat{E}} \sigma(t)$ iff
$t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and $\sigma(\tilde{s}_{\pi(i)}) =_{\hat{E}} \sigma(\tilde{t_i})$ for some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

c) Let $s \equiv x$. If $\sigma(s) =_{\hat{E}} \sigma(t)$ then either $t \equiv x$ or $x \notin \mathrm{Var}(t)$.

Proof: This follows from Lemma 4.1.

Again, based on this lemma one can design a polynomial non-deterministic algorithm for the E-unifiability problem. And one can design a relatively efficient algorithm for computing a $\mathrm{CSU}_{\hat{E}}(s,t)$ for any two stratified terms s, t and any set E of permutative equations.

Theorem 4.3
The problem to decide for stratified terms s, t whether there is a σ such that $\sigma(s) =_{\hat{E}} \sigma(t)$ is NP-complete. □

Definition 4.3 poses a slightly different unification problem: Compute $\Sigma(s,t) = \{mgu(s', t') \mid (s', t') \in Um(s) \times Um(t)\}$. The next lemma helps to solve this problem. We use c_f^{ε} to denote either c_f or $\hat{c}_f$.

Lemma 4.4
Let $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ be stratified and $\Sigma^*(s,t) = \{\sigma \mid \sigma(s') \equiv \sigma(t'), (s', t') \in Um(s) \times Um(t)\}$.

a If $s \equiv f(s_1, \ldots, s_k)$ then
$\sigma \in \Sigma^*(s,t)$ iff
$t \equiv f(t_1, \ldots, t_k)$ and $\sigma \in \Sigma^*(s_i, t_i)$ for all $i = 1, \ldots, k$ or
$t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and for some $\pi \in G_{\hat{f}}$: $t^\pi \equiv \hat{c}_f[\tilde{t}_{\pi(1)}, \ldots, \tilde{t}_{\pi(l)}] \equiv \hat{f}(t_1^\pi, \ldots, t_k^\pi)$ and $\sigma \in \Sigma^*(s_i, t_i^\pi)$ for all $i = 1, \ldots, k$.

b If $s \equiv \hat{c}_f[\tilde{s_1}, \ldots, \tilde{s_l}]$ then
$\sigma \in \Sigma^*(s,t)$ iff
$t \equiv c_f^{\varepsilon}[\tilde{t_1}, \ldots, \tilde{t_l}]$ and $\sigma \in \Sigma^*(\tilde{s}_{\pi(i)}, \tilde{t_i})$ for some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$ or
$t \equiv f(t_1, \ldots, t_k)$ and for some $\pi \in G_{\hat{f}}$: $s^\pi \equiv \hat{c}_f[\tilde{s}_{\pi(1)}, \ldots, \tilde{s}_{\pi(l)}] \equiv \hat{f}(s_1^\pi, \ldots, s_k^\pi)$ and $\sigma \in \Sigma^*(s_i^\pi, t_i)$ for all $i = 1, \ldots, k$.

c If $s \equiv x$ and $\sigma \in \Sigma^*(s,t)$ then $t \equiv x$ or $x \notin \mathrm{Var}(t)$. □

Using this lemma and the techniques used for algorithm *Unify* in section 3 one can easily design an efficient algorithm to compute a system of constraints in solved form that define exactly the $\sigma \in \Sigma(s,t)$.

Theorem 4.4
The unifiability problem modulo $\hat{E}$ for permutative axioms E is NP-complete. $\square$

We have discussed in [AP01] several other problems that need to be solved for computing modulo $\hat{E}$. Here we study two of them.

Definition 4.4 (Subset modulo $\hat{E}$)
Input: $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ stratified.
Question: Is $Um(s) \subseteq Um(t)$?

Definition 4.5 (IntSectMty modulo $\hat{E}$)
Input: $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ stratified.
Question: Is $Um(s) \cap Um(t)$ not empty?

We give the basic lemmas to solve these problems.

Lemma 4.5
Let $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ be stratified.

a) If $s \equiv f(s_1, \ldots, s_k)$ then
Subset(s, t) iff
$t \equiv f(t_1, \ldots, t_k)$ and *Subset*(s_i, t_i) for all $i = 1, \ldots, k$ or
$s \equiv c_f[\tilde{s_1}, \ldots, \tilde{s_l}]$, $t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and *Subset*$(\tilde{s}_{\pi(i)}, \tilde{t_i})$ for some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

b) If $s \equiv \hat{c}_f[\tilde{s_1}, \ldots, \tilde{s_l}]$ then
Subset(s, t) iff
$t \equiv c^{\varepsilon}_f[\tilde{s_1}, \ldots, \tilde{s_l}]$ and *Subset*$(\tilde{s}_{\pi(i)}, \tilde{t_i})$ for all $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

c) If $s \equiv x$ then *Subset*(s, t) iff $t \equiv x$. $\square$

Lemma 4.6
Let $s, t \in \mathrm{Term}(F \cup \hat{F}, V)$ be stratified.

a) If $s \equiv f(t_1, \ldots, t_k)$ then
IntSectMty(s, t) iff
$t \equiv f(t_1, \ldots, t_k)$ and *IntSectMty*(s_i, t_i) for all $i = 1, \ldots, k$ or
$s \equiv c_f[\tilde{s_1}, \ldots, \tilde{s_l}]$, $t \equiv \hat{c}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and *IntSectMty*$(\tilde{s}_{\pi(i)}, \tilde{t_i})$ for some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

b) If $s \equiv \hat{c}_f[\tilde{s_1}, \ldots, \tilde{s_l}]$ then
IntSectMty(s, t) iff
$t \equiv c^{\varepsilon}_f[\tilde{t_1}, \ldots, \tilde{t_l}]$ and *IntSectMty*$(\tilde{s}_{\pi(i)}, \tilde{t_i})$ for some $\pi \in G_{\hat{f}}$ and all $i = 1, \ldots, l$.

c) If $s \equiv x$ then *IntSectMty*(s, t) iff $t \equiv x$.

These problems can be solved similar to equality testing modulo $\hat{E}$: We have to store the results of the tests whether $Subset(s', t')$ and $IntSectMty(s', t')$ holds, respectively, for $(s', t') \in Sub(s) \times Sub(t)$ whenever these tests are performed. This gives

Theorem 4.5
For any fixed set E of permutative axioms the problems *Subset* modulo $\hat{E}$ and *IntSectMty* modulo $\hat{E}$ can be solved in polynomial time. $\square$

References

[AP01] Jürgen Avenhaus and David Plaisted. General algorithms for permutations in equational inference. *Journal of Automated Reasoning*, (26):223–268, 2001.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme (in German)*. Springer, 1995.

[BG01] Leo Bachmair and Harald Ganzinger. *Resolution theorem proving, in [RV01]*, volume 1, pages 19–99. Elsevier, 2001.

[BKN85] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. In J.P. Jouannaud, editor, *Int. Conf. on Term Rewriting and Applications*, number 202 in LNCS, pages 417–429. Springer, 1985.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[BS01] Franz Baader and Wayne Snyder. *Unification theory, in [RV01]*, volume 1, pages 445–533. Elsevier, 2001.

[Cam02] Peter Cameron. *Private communication*. 2002.

[CLR96] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to algorithms*. MIT Press, 1996.

[dlTE03] Thierry Boy de la Tour and Mnacho Echenim. Np-completeness results for deductive problems on stratified terms. In Moshe Y. Vardi and Andrei Voronkov, editors, *10th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, number 2850 in LNCS, pages 317–331. Springer, 2003.

[FHL80] M. Furst, J. Hopcroft, and E. Luks. Polynomial time algorithms for permutation groups. In *Proc. 21st IEEE Symp. on Foundations of Computer Science*, pages 36–41, 1980.

[HL02] Thomas Hillenbrand and Bernd Löchner. The next WaldMeister loop. In A. Voronkov, editor, *Int. Conf. on Automated Deduction*, number 2392 in LNAI, pages 486–500. Springer, 2002.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.J. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[NR01] Robert Nieuwenhuis and Albert Rubio. *Paramodulation-based theorem proving, in [RV01]*, volume 1, pages 371–443. Elsevier, 2001.

[Pla95] David Plaisted. *Private communication*. 1995.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.