Semi-unification*

Deepak Kapur  
*State University of New York at Albany, Computer Science Department, Albany, NY 12222, USA*  

David Musser  
*Rensselaer Polytechnic Institute, Computer Science Department, Troy, NY 12181, USA*  

Paliath Narendran  
*General Electric Company, Corporate Research and Development, Schenectady, NY 12345, USA*  

Jonathan Stillman**  
*State University of New York at Albany, Computer Science Department, Albany, NY 12222, USA*

Abstract


Semi-unification is a generalization of both matching and ordinary unification: for a given pair of terms \( s \) and \( t \), two substitutions \( \rho \) and \( \sigma \) are sought such that \( \rho(\sigma(s)) = \sigma(t) \). Semi-unifiability can be used as a check for non-termination of a rewrite rule, but constructing a correct semi-unification algorithm has been an elusive goal; for example, an algorithm given by Purdom in his RTA-87 paper was incorrect. This paper presents a decision procedure for semi-unification based on techniques similar to those used in the Knuth-Bendix completion procedure. When its inputs are semi-unifiable, the procedure yields a canonical term-rewriting system from which substitutions \( \rho \) and \( \sigma \) are easily extracted. Though exponential in its computing time, the decision procedure can be improved to a polynomial-time algorithm, as will be shown.

1. Introduction

Term rewriting systems have many uses, including formula simplification, deciding equations, program transformation, program verification, and checking consistency and completeness of abstract data type specifications. In these and many other

* This paper was presented at the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science (Pune, 1988) and selected for publication by K.V. Nori.
** Some of the results reported here are a partial fulfillment of the Ph.D. requirements of the last author, and will be part of his dissertation.

0304-3975/91/$03.50 © 1991—Elsevier Science Publishers B.V.
applications, it is necessary that a set of rewrite rules have the property of uniform termination; i.e., every sequence of rewrites using the rules is finite (ends with a term to which no rule applies). Although uniform termination is undecidable in general [6], a number of sufficient conditions have been proposed in the literature [3] and checks for these conditions have been implemented in theorem proving systems such as RRL [9]. Here we consider the dual problem, i.e., that of exhibiting a sufficient condition for nontermination. One condition that has been proposed [12] is based on left-unification, a generalization of matching and unification. This is a local condition on a single rule and is useful as a first check on rules proposed to be included in a term rewriting system as they may be generated, for example, by the Knuth-Bendix procedure [10, 7] for completing a given set of rules into a decision procedure for an equational theory. A rule that satisfies the condition is nonterminating by itself; thus, there is no point in attempting to prove uniform termination of a term rewriting system that includes it. A typical situation in a rewrite rule based theorem prover such as RRL is that if a newly generated equation cannot be oriented under a termination ordering used to orient previously considered equations, then an extension to the ordering is attempted under which the new equation can be oriented. In case such an extension cannot be guessed easily, it might be useful to perform the check for left-unification of both the orientations of the new equation before the user has to backtrack to a previous decision point where the ordering was last extended and try a different extension.

The purpose of this paper is to present a left-unification algorithm, outline a proof of its correctness (an algorithm previously given by Purdom [15] was incorrect, as we will show), and to improve the efficiency of the algorithm from exponential to polynomial time. First, we return to the motivation for generalizing unification.

There are some obvious conditions on a rule that are sufficient for it to be nonterminating. For example, if the left-hand side of a proposed rule matches a nonvariable subterm of the right-hand side, as in

\[ f(x) \rightarrow g(x, a, f(h(x))) \]

then the rule is nonterminating (in these examples, \( a, b, c \) are constants and \( x, y, z \) are variables). Another sufficient condition for nontermination is that the left-hand side unifies with a nonvariable subterm, as in

\[ f(a, x) \rightarrow g(f(x, a)) \]

where the substitution \( \theta = \{ x \leftarrow a \} \) unifies the left side with the subterm \( f(x, a) \) of the right side.

However, in an example like

\[ f(g(x), a, y) \rightarrow f(g(g(x)), y, a) \]  \hspace{1cm} (1)

the left-hand side neither matches nor unifies with any subterm of the right-hand side, but the rule is nonterminating, e.g.,

\[ f(g(x), a, a) \rightarrow f(g(g(x)), a, a) \rightarrow f(g(g(g(x))), a, a) \rightarrow \cdots. \]
A more comprehensive sufficient condition for nontermination is based on the following:

**Definition 1.1.** Given an ordered pair of terms \( s \) and \( t \), we say \( s \) left-unifies with \( t \) if there is a pair of substitutions \( \rho \) and \( \sigma \) such that \( \rho(\sigma(s)) = \sigma(t) \).

For example, with the rule (1), the left-hand side left-unifies with the right-hand side, with \( \sigma = \{ y \leftarrow a \} \) and \( \rho = \{ x \leftarrow g(x) \} \).

Left-unification properly generalizes both matching (the case when \( \sigma \) is the identity substitution) and ordinary unification (when \( \rho \) is the identity substitution). Of course, one could define a corresponding notion of right-unification; we use the term *semi-unification* to signify either left- or right-unification. It is obvious that two terms may be semi-unifiable despite their being not unifiable; the pair of terms \( x \) and \( f(x) \) is a simple example.

We have the following simple theorem, which says that left-unification can be used as a basis of a sufficient condition for nontermination.

**Theorem 1.2.** If the left-hand side of a rewrite rule left-unifies with a nonvariable subterm of the right-hand side, the rule is nonterminating.

**Proof.** Suppose \( L \rightarrow R \) is the rule and \( \rho(\sigma(L)) = \sigma(R') \) for some nonvariable subterm \( R' \) of \( R \). Then

\[
\rho(\sigma(L)) \rightarrow \rho(\sigma(R))
\]

and within \( \rho(\sigma(R)) \) the subterm \( \rho(\sigma(R')) \) occurs, which is equal to \( \rho(\rho(\sigma(L))) \) and therefore can be rewritten to \( \rho(\rho(\sigma(R))) \), and so on. \( \square \)

The condition of Theorem 1.2 is not necessary for nontermination; a simple example (given in [3]) is

\[
f(g(x)) \rightarrow g(g(f(f(x))))
\]

which is nonterminating (consider the term \( f(f(g(a))) \), for example) even though the \( f(g(x)) \) fails to left-unify with any nonvariable subterm of \( g(g(f(f(x)))) \). Nevertheless, many nonterminating rules found in practice do satisfy the condition.

Musser and Lankford defined the notion of left-unification and discussed its use as a sufficient condition for nontermination based on the above theorem in a privately circulated 1978 memo [12]. Attempts were made to use the test in the Affirm program verification system [16] as part of an implementation of the Knuth–Bendix completion procedure, but a fully correct left-unification algorithm was not known at the time. Dershowitz cited [12] and discussed the problem as part of his comprehensive survey of rewrite-rule termination and nontermination in [3]. Purdom [15] studied the problem, presented a left-unification algorithm and a generalization of Theorem 1.2, and reported extensive positive experience with its use in testing for nontermination of rules in the Knuth–Bendix completion procedure. Unfortunately, however,
Purdom's algorithm was not correct; a counter-example is the pair of terms \( s = f(h(y), x) \) and \( t = f(x, h(h(y))) \), which will be discussed later.

Another practical application of left-unification which has been suggested recently arises in the area of type inference in extensions of the Milner Calculus (the typed \( \lambda \)-calculus which underlies the programming languages ML, Miranda, and several other strongly typed polymorphic functional languages). This is discussed more fully in [4], where it is shown that one can reduce the polymorphic type inference problem for an extension of the Milner Calculus to a generalized left-unification problem.

In this paper we present two algorithms for left-unification and outline proofs of their correctness. (Obviously a left-unification algorithm can also be used as a right-unification algorithm merely by reversing the order of the pair of input terms \( s \) and \( t \).) The first algorithm is presented mainly for expository purposes, as it takes exponential computing time in the worst case; we show that the second, more complex algorithm has a polynomial time bound.

The first algorithm is presented in two parts: a decision procedure for semi-unifiability, and an algorithm for extracting the semi-unifying substitutions from information computed by the decision procedure. Note that the application to testing for nontermination of a rewrite rule only requires the result of the decision procedure.

2. A decision procedure

We show how to construct, given an instance of semi-unification, an equational algebra such that semi-unifiability of the original problem is equivalent to a certain syntactic condition on the algebra. The equational algebra can be said to "model" the semi-unification problem. The construction of this equational algebra is outlined below.

Let \( s \) and \( t \) be the input terms and \( \rho \) and \( \sigma \) be the substitutions that we are looking for, such that \( \rho(\sigma(s)) = \sigma(t) \). Let \( V \) stand for the set of variables occurring in \( s \) and \( t \) and \( F \) stand for the set of function symbols in \( s \) and \( t \). For each variable \( x \) in \( V \), we have a constant symbol \( s_x \) which represents \( \sigma(x) \). Let \( \theta \) be the substitution that maps every \( x \) in \( V \) to the corresponding \( s_x \); i.e.,

\[
\theta = \{ x \leftarrow s_x \mid x \in V \}.
\]

Note that since \( \theta \) is a bijection, \( \theta^{-1} \) is also a bijection.

The equational algebra \( E(s, t) \) consists of the equation

\[
\rho(\theta(s)) = \theta(t),
\]

along with the following equations which expresses the fact that \( \rho \) "distributes" over the function symbols; thus for each \( f \) in \( F \), we have

\[
\rho(f(x_1, \ldots, x_n)) = f(\rho(x_1), \ldots, \rho(x_n))
\]

where \( x_1, \ldots, x_n \) are variables and \( n \) is the arity of \( f \).
We also have some meta-rules expressing the cancellativity of functions in $F$. That is,
\[ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \text{ implies } x_1 = y_1, \ldots, x_n = y_n \]
for every function $f$ in $F$, where $n = \text{arity}(f)$. These meta-rules are obtained from the fact that in order to unify (or match) two terms with the same outermost function symbol, their respective arguments must be unified (matched).

It can be seen easily that this equational theory is essentially a collection of properties that $\rho$ and $\sigma$ have to satisfy in order for them to semi-unify $s$ and $t$. For instance, the “distribution” equations express the fact that the substitutions are applied only to variables.

Let $=_{E}$ stand for the congruence generated by $E(s, t)$.

We can now give an algorithm for the semi-unification problem, which we divide into two parts:

Algorithm A-1: This algorithm decides whether two given terms can be semi-unified or not, and produces a canonical rewriting system [7] if the given terms are semi-unifiable.

Algorithm A-2: This algorithm extracts the matching substitution $\rho$ and the unifying substitution $\sigma$ from the rewriting system produced by Algorithm A-1.

This algorithm is simple at the cost of being exponential in the size of the input terms. In the next section, a modification of Algorithm A-1 is discussed which is shown to run in a number of steps polynomial in the size of the input terms.

**Algorithm A-1 (For deciding semi-unifiability):**

Step 1. Start with the equation $\rho(\sigma(s)) = \sigma(t)$.

Step 2. Apply the distributivity equations and the meta-rules of cancellativity discussed above to obtain equations such that at least one side of every equation is either of the form $s_x$ or $\rho^i(s_x)$, where $x$ is a variable in $V$. In this process, if either of the following two conditions arises, then we stop and report failure.

1. Root conflict, i.e., an equation $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ is encountered where $f$ and $g$ are distinct function symbols.
2. An equation $\rho^i s_x = f(\ldots \rho^{i+1} s_x \ldots)$ is encountered, where by $f(\ldots \rho^{i+1} s_x \ldots)$ we mean a term with $f$ as the top-level symbol and $\rho^{i+1} s_x$ as a subterm. This situation is similar to the “occurs check” in ordinary unification.

It is shown later that the above two conditions are necessary and sufficient for terms $s$ and $t$ to be not semi-unifiable.

Step 3. The equations derived in Step 2 are oriented into rewrite rules as follows:

1. A total ordering $>$ is defined on the set $S = \{s_x | x \in V\}$.
2. A term containing symbols from $F$ is always considered lower than one that does not.
3. Terms containing $\rho$ and symbols from $S$ are considered as strings and compared lexicographically from right to left using $>$. 
It is shown below that terms $s$ and $t$ are semi-unifiable if and only if there are no root conflicts in obtaining equations from $\rho(\sigma(s))$ and $\sigma(t)$ and the rewrite rules obtained from them are always terminating.

Step 4. For each rule, reduce each side (if reducible) by a single step of rewriting by other rules. Replace the rule by the new equation thus obtained and go to Step 2. If no rule can be rewritten any further, report semi-unifiability and return the rewriting system $R(s, t)$.

Note that this procedure has some resemblance to a procedure described by Plaisted in [14] in the way nontermination of rewrite rules is detected.

It can be shown that Algorithm A-1 will always terminate either reporting failure or semi-unifiability; in the latter case, the rewrite system $R(s, t)$ is generated which is a reduced canonical rewriting system (modulo cancellativity) and every rule in it is of the form

$$\rho^i s \rightarrow t$$

where $i \geq 0$ and $t$ is a ground term.

Algorithm A-2 below is a method for determining $\rho$ and $\sigma$ from $R(s, t)$ in the case Algorithm A-1 decides that $s$ and $t$ are semi-unifiable. Before discussing Algorithm A-2, we illustrate Algorithm A-1 on several examples.

**Example 2.1.** Consider the terms $s = g(f(x, y), f(y, z))$ and $t = g(z, x)$. After distributing $\rho$ over function symbols in equations obtained from $s$ and $t$ and applying cancellativity (Step 2), we have

$$f(\rho(s_x), \rho(s_y)) = s_z, \quad f(\rho(s_y), \rho(s_z)) = s_x.$$  

From these equations, we obtain the following rewrite rules (Step 3):

$$s_x \rightarrow f(\rho(s_x), \rho(s_y)), \quad s_x \rightarrow f(\rho(s_y), \rho(s_z)).$$

After the first rule is used to rewrite the second rule (Step 4) and after “pushing $\rho$’s down” using distributivity of $\rho$, we obtain the equation

$$s_x = f(\rho(s_x), f(\rho^2(s_x), \rho^2(s_y))),$$

which implies that $s$ and $t$ are not semi-unifiable (condition (2) in Step 2).

The following example illustrates non-semi-unifiability due to a root conflict.

**Example 2.2.** Consider the terms $i(f(x), f(f(y)), g(x, v), g(h(y), f(w)))$ and $i(v, v, z, z)$. After distributing $\rho$ over function symbols in equations obtained from these terms and applying cancellativity (Step 2), we have

$$f(\rho(s_x)) = s_v, \quad f(f(\rho(s_y))) = s_v,$$

$$g(\rho(s_x), \rho(s_v)) = s_z, \quad g(h(\rho(s_y)), f(\rho(s_w))) = s_z.$$
In Step 3, the above equations are oriented as rewrite rules as follows:

\[ s_x \rightarrow f(\rho(s_x)), \quad s_y \rightarrow f(f(\rho(s_y))), \]
\[ s_z \rightarrow g(\rho(s_z), \rho(s_y)), \quad s_z \rightarrow g(h(\rho(s_y)), f(\rho(s_w))). \]

In Step 4, the first rule is used to rewrite the second rule and the third rule is used to rewrite the fourth rule, which give the following equations replacing the second and fourth rules:

\[ f(\rho(s_x)) = f(f(\rho(s_y))), \quad g(\rho(s_z), \rho(s_y)) = g(h(\rho(s_y)), f(\rho(s_w))). \]

In Step 2, using cancellativity, we get new equations:

\[ \rho(s_x) = f(\rho(s_y)), \quad \rho(s_z) = h(\rho(s_y)), \quad \rho(s_w) = f(\rho(s_w)). \]

From these equations, we get new rules in Step 3:

\[ \rho(s_x) \rightarrow f(\rho(s_y)), \quad \rho(s_z) \rightarrow h(\rho(s_y)), \quad \rho(s_w) \rightarrow f(\rho(s_w)). \]

The second rule above is rewritten using the first rule to give the following equation in Step 4:

\[ f(\rho(s_x)) = h(\rho(s_y)). \]

Then, in Step 2, the root conflict is detected and non-semi-unifiability is declared.

The following example illustrates the algorithm for the case when the given terms are semi-unifiable.

**Example 2.3.** Consider the terms \( g(f(x), f(z)) \) and \( g(y, x) \). After distributing \( \rho \) over function symbols in equations obtained from \( s \) and \( t \) and applying cancellativity in Step 2, we have

\[ f(\rho(s_x)) = s_y, \quad f(\rho(s_z)) = s_x. \]

In Step 3, we get the following rewrite rules:

\[ s_x \rightarrow f(\rho(s_z)), \quad s_y \rightarrow f(\rho(s_x)). \]

After the first rule is used to rewrite the second rule in Step 4, we obtain a new equation which, after distributing \( \rho \) in Step 2, is

\[ s_y = f(f(\rho^2(s_z))) \]

In Step 3, this equation gives a new rule, replacing the second rule. We thus have

\[ s_x \rightarrow f(\rho(s_z)), \quad s_y \rightarrow f(f(\rho^2(s_z))). \]

These rewrite rules cannot be further rewritten, thus giving a reduced canonical system. Algorithm A-1 reports that the given terms are semi-unifiable.
3. Substitution extraction algorithm

We now discuss how to generate $\rho$ and $\sigma$ from a reduced canonical system $R(s, t)$
when Algorithm A-1 reports semi-unifiability of terms $s$ and $t$.

The left-hand side of every rule in $R(s, t)$ is of the form $\rho^i(s_x)$, where $x$ is a
variable in $V$ and $i \geq 0$. We introduce new variables, and thereby new symbols of
the form $s_u$ to get rid of the $\rho$’s. For example, if $\rho(s_x)$ occurs in a rule we will
uniformly replace it with some new symbol $s_u$, where $u$ is a new variable not in $V$.
Clearly there cannot be a rule with $s_x$ as its left-hand side, for otherwise $\rho s_x$ would
have been reduced further. We take $\sigma(x) = x$ and $\rho(x) = \sigma(u)$, after $\sigma(u)$ is computed
by repeating the process. On the other hand, if $s_x$ does appear as a left-hand side
for some variable $x$, and the right-hand side does not contain $\rho$’s, then applying
$\theta^{-1}$ to the right-hand side gives us $\sigma(x)$. It can be seen that this process will terminate
giving us the semi-unifiers.

Algorithm A-2 (For extracting $\rho$ and $\sigma$ from $R(s, t)$):

Step 1. Initialize: $\rho = \emptyset$; $\sigma = \emptyset$; $V = V$.

Step 2. Process the right-hand sides:

For all variables $x$ in $V$ do
if $\rho s_x$ appears in an rhs,
   replace $\rho s_x$ everywhere with $s_u$ where $u$ is a new variable;
   $\rho := \rho \cup \{x \leftarrow u\}$;
   $V := V \cup \{u\}$.

Repeat this until there are no $\rho$’s on the right-hand side of any rule.

Step 3. Compute $\sigma$:

For every rule in which $\rho$ does not occur do
For all variables $x$ in $V$ do
if $s_x$ occurs in the rhs
   then $\sigma(x) = x$
else $\sigma(x) = \theta^{-1}(r)$ where $r$ is the rhs.

Recall that $\theta^{-1}$ merely replaces $s_x$ by $y$.

Step 4. Compute the remaining part of $\rho$:

For all rules of the form $\rho s_x \rightarrow t$ do
Introduce new variables $u_1, \ldots, u_{i-1}$ and make
   $\rho(x) = u_1$,
   $\rho(u_1) = u_2$,
   $\vdots$
   $\rho(u_{j-1}) = u_j$,
   $\vdots$
   $\rho(u_{i-1}) = \theta^{-1}(t)$.

After these steps, we have obtained $\rho$ and $\sigma$. Note that the above algorithm may
introduce unnecessary new variables. In particular, if no function symbol in $F$ is
used in a $R(s, t)$, then we do not need to introduce any new variable to get the semi-unifiers.

Let us illustrate Algorithm A-2 first on the result of Algorithm A-1 on Example 2.3 above.

**Example 2.3 (continued).** The rewrite system obtained as the result of Algorithm A-1 on Example 2.3 is

$$s_x \rightarrow f(p(s_z)), \quad s_y \rightarrow f(f(p^2(s_z))).$$

In Step 2 above, we replace $p(s_z)$ by $s_u$ and $p(s_u) = p^2(s_z)$ by $s_u$. This makes $\rho = \{ z \leftarrow u_1, u_1 \leftarrow u_2 \}$. In Step 3, we obtain $\sigma = \{ x \leftarrow f(u_1), y \leftarrow f(f(u_2)) \}$.

The following example illustrates both parts of the algorithm. This is also a counter-example to Purdom's algorithm.

**Example 3.1.** Consider the terms $s = f(h(y), x)$ and $t = f(x, h(h(y)))$. In Step 2 of Algorithm A-1, after pushing $\rho$ down and applying cancellativity, we have

$$h(p(s_z)) = s_x, \quad p(s_x) = h(s_y)$$

From these equations, we get the rewrite rules in Step 3,

$$s_x \rightarrow h(p(s_z)), \quad p(s_x) \rightarrow h(s_y).$$

Since the former reduces the latter (Step 4), we replace the second rule by the new equation

$$h(p(p(s_z))) = h(s_y),$$

and, in Step 2, after cancelling $h$, we get the rewrite rule in Step 3,

$$p(p(s_z)) \rightarrow h(s_y).$$

Both of these rules are already reduced thus giving a reduced canonical rewrite system:

$$s_x \rightarrow h(p(s_z)), \quad p(p(s_z)) \rightarrow h(s_y).$$

Using Algorithm A-2, we can extract the solution as follows. In Step 2, we replace $p(s_z)$ in the right-hand side of the first rule by $s_u$, and make $\rho = \{ y \leftarrow u \}$. In Step 3, we get $\sigma = \{ x \leftarrow h(u) \}$. In Step 4, $\rho$ is extended to include $u \leftarrow h(y)$. The following are the semi-unifiers.

$$\sigma = \{ x \leftarrow h(u) \} \quad \text{and} \quad \rho = \{ y \leftarrow u, u \leftarrow h(y) \}.$$

Purdom's algorithm [15] fails on this example, since it first adds $\{ x \leftarrow h(y) \}$ to $\sigma$. This would result in terms $f(h(y), h(y))$ and $f(h(y), h(h(y)))$ which are not semi-unifiable.
The reader can easily observe that in the worst case, this algorithm may take exponentially many steps to obtain a reduced canonical rewriting system in Algorithm A-1. In addition, the number of occurrences of $p$ may increase exponentially. Consider, for example the case where one of the terms is "right-heavy", and the other is "left-heavy" (e.g., let $s = f(x_1, f(x_2, \ldots , f(x_{n-1}, x_n) \ldots ))$, and $t = f(f(\ldots f(x_n, x_{n-1}) \ldots ))(x_2), x_1)$). Here the number of occurrences of $p$ in the canonical system generated by Algorithm A-1 is exponential in the original presentation. In a later section, we discuss a modification of Algorithm A-1 which can decide semi-unifiability in polynomial time. In the following section we give the necessary and sufficient conditions for semi-unifiability which serve as the basis of correctness of the algorithm discussed above.

4. Underlying theory

The equational congruence $E(s, t)$ is said to have a root conflict if and only if there exist distinct functions $f$ and $g$ in $F$ such that $f(t_1, \ldots , t_m)$ is congruent modulo $E$ to $g(r_1, \ldots , r_n)$, where $t_1, \ldots , t_m, r_1, \ldots , r_n$ are terms and $m$ and $n$ are the arities of $f$ and $g$ respectively.

**Theorem 4.1.** Two terms $s$ and $t$ are not semi-unifiable if and only if $E = E$, the congruence generated by $E(s, t)$, satisfies either of the following properties:

1. $E(s, t)$ has a root conflict.
2. There exists a variable $x$ in $V$, a function $f$ in $F$, and nonnegative integers $i, j$ such that

$$p^i s_x = E f(\ldots p^{i+j} s_x \ldots ).$$

**Proof.** The "if" case is straightforward, since (a) no substitution on the variables can change the top-level symbol of a term, and (b) no substitution can "shrink" a term. We prove the "only if" part by construction, i.e., by showing how $p$ and $\sigma$ can be obtained if conditions (1) and (2) are not satisfied. The basic idea is this: if (1) and (2) are not satisfied, we can construct a reduced canonical term rewriting system equivalent to $E(s, t)$ using Algorithm A-1 discussed above and from which $p$ and $\sigma$ can be "extracted" from Algorithm A-2 above.

The theorem follows from the following two lemmas.

Define the weight of a term $r$, denoted by $w(r)$, as the number of function symbols from $F$ in it. It can be shown that the second condition specified in the statement of the above theorem is equivalent to that of having two terms of different weights that are congruent modulo $E(s, t)$ such that one is homeomorphically embedded in the other, provided the first condition does not hold (i.e., there are no root conflicts).
A term $s$ is \textit{homeomorphically embedded} in a term $t$, written $s \equiv t$, if

1. $s$ and $t$ are identical, or
2. $t$ is $g(t_1, \ldots, t_n)$ and $s \equiv t_i$ for some $i$, or
3. $s$ is $f(s_1, \ldots, s_m)$ and $t$ is $f(t_1, \ldots, t_n)$ and $\vec{s} = (s_1, \ldots, s_m) \equiv \vec{t} = (t_1, \ldots, t_n)$ where $\vec{s} = (s_1, \ldots, s_m) \equiv \vec{t} = (t_1, \ldots, t_n)$ iff (length($\vec{s}$) $\leq$ length($\vec{t}$)) and
4. $\vec{s} = ()$, the empty list, or
5. $s_1 \equiv t_1$ and $(s_2, \ldots, s_m) \equiv (t_2, \ldots, t_n)$, or
6. $\vec{s} \equiv (t_2, \ldots, t_n)$.

\textbf{Lemma 4.2.} Let $s, t$ be two terms and $E(s, t)$ be as defined above. Further, assume that $E(s, t)$ has no root conflicts. Then the following two conditions are equivalent.

(a) There exists a variable $x$ in $V$, a function $f$ in $F$, and nonnegative integers $i, j$ such that

$$p^i s_x = E f(\ldots p^{i+j} s_x \ldots),$$

(b) There exist terms $t_1$ and $t_2$ such that $w(t_1) < w(t_2)$, $t_1 =_E t_2$, and $t_1$ is homeomorphically embedded in $t_2$.

\textbf{Proof.} Condition (b) can be easily seen to be implied by (a). Going the other way requires use of cancellativity. \qed

\textbf{Lemma 4.3.} Let $s, t$ be two terms and $E(s, t)$ be as defined above. Further, assume that $E(s, t)$ has no root conflicts. Then $E(s, t)$ has an infinite congruence class if and only if either of the following two conditions hold.

(a) There exist nonnegative integers $i, j$ such that $p^i s_x$ is congruent to $p^{i+j} s_x$ for some $x \in V$.

(b) There exist terms $t_1$ and $t_2$ such that $w(t_1) < w(t_2)$, $t_1$ is congruent to $t_2$, and $t_1$ is homeomorphically embedded in $t_2$.

The proofs are straightforward and thus omitted.

\textit{Towards a polynomial-time algorithm}

One of the major problems with the algorithm given in the previous section is the proliferation of $\rho$'s in the suggested completion procedure. (There are other problems as well, such as expansion in size while getting a \textit{reduced} canonical system. This is similar to the situation in standard unification.) Our way around this is to establish a notion of cancellativity for $\rho$ as well, so that whenever we get an equation of the form

$$\rho t_1 = \rho t_2,$$

we can immediately cancel the $\rho$'s and get $t_1 = t_2$. 
We thus augment the equational theory \( E(s, t) \) with a meta-rule for the cancelativity of \( \rho \):

\[
\rho t_1 = \rho t_2 \text{ implies } t_1 = t_2.
\]

We refer to this modified equational theory by \( E'(s, t) \) and abbreviate the congruence it generates by \( =_{E'} \).

The following key theorem justifies our making \( \rho \) cancellative as far as checking for semi-unifiability is concerned.

**Theorem 4.4.** For all \( s, t, t_1 \) and \( t_2 \), \( t_1 \) and \( t_2 \) are congruent modulo \( E'(s, t) \) if and only if there exists a nonnegative integer \( i \) such that \( \rho^i t_1 \) and \( \rho^i t_2 \) are congruent modulo \( E(s, t) \).

**Proof.** The "if" part is trivial. The "only if" part is proved by induction on the number of proof steps involved in showing that \( t_1 \) and \( t_2 \) are congruent modulo \( E'(s, t) \), where a proof step is either (i) a cancellation step, or (ii) a replacement using an equation proved earlier. \( \square \)

5. A polynomial algorithm for semi-unifiability

In this section we describe a polynomial algorithm for deciding semi-unifiability. We begin with a brief discussion of several key issues.

First, the data structure we use is a graph representation of the rewriting system described previously. We construct a reduced directed acyclic graph \(^1\)(common subexpressions occur uniquely) corresponding to the two terms under consideration. (The algorithm we describe is closely related to the unification algorithm described by Paterson and Wegman in [13], the main difference being that we must deal with the \( \rho \) function. As a result of this, our arcs describing equivalences have arguments and a simple check for acyclicity is insufficient as a test for semi-unifiability. To see the similarity between the algorithm to be presented and the Paterson–Wegman unification algorithm, one may consider the special case of our algorithm when the initial call has the cost field of the arc set equal to 0.) Initially, these will be the only arcs in the graph, but the arcs that are added by the procedures below will be considered to be of a different type: the initial arcs simply denote the paths from a function symbol to its arguments; the second type is more critical to the algorithm, and will be denoted as tuples \((\text{tnode}, \text{hnode}, \text{cost}, \text{dir})\), where \( \text{tnode} \) and \( \text{hnode} \) represent the labels associated with the corresponding nodes, \( \text{cost} \) is a natural

\(^1\) It should be noted that as the algorithm proceeds, the graph may lose its acyclicity. This is not critical to the algorithm; the main advantage to starting with a reduced directed acyclic graph is that variables (actually the constants \( s \), representing them) occur uniquely.
number and dir (for direction) is either $\rightarrow$ or $\leftarrow$, and the arc is from tnode to hnode. Thus the arc $(tnode, hnode, i, \rightarrow)$ corresponds to the rule $\rho'tnode \rightarrow hnode$, and the arc $(tnode, hnode, i, \leftarrow)$ corresponds to the rule $tnode \rightarrow \rho'hnode$.

Next, we must avoid the exponential number of occurrences of $\rho$ that may occur as the computation progresses. This is easily addressed by storing a bit vector representing the number of occurrences of $\rho$ at a given term. Thus, although the number of $\rho$'s may become exponential we can always store this information using a small number of bits.

Finally, we must show that the algorithm does not allow an exponential number of distinct rules to be introduced. This is done by showing that after a polynomial number of iterations (at each of which the costs on arcs may as much as double) the introduction of each new arc between two nodes in the graph will have its cost being at most half of any previously introduced arc between the same two nodes. Thus the number of bits needed to represent the new cost is at least one less than needed previously. Since the number of bits needed can be bounded from above by a polynomial, the algorithm is guaranteed to terminate in polynomial time. This is discussed in more detail after the algorithm is presented.

In the following we assume that $\mathcal{V}$ represents the set of constants introduced corresponding to the variables occurring in the original terms.

### 5.1. Algorithm B

We describe the algorithm below: (for left-unification, the initial call is to Propagate with the arc $(\text{root}(t1), \text{root}(t2), 1, \rightarrow)$; the call for right-unification is to Propagate with the arc $(\text{root}(t1), \text{root}(t2), 1, \leftarrow)$). Once the procedure below terminates (when there are no more calls to propagate an arc, assuming no function-symbol conflicts were detected), it is a straightforward matter to compute whether there is an $s_x$ such that

$$\rho's_x = ef(\ldots \rho'^{s_x} \ldots)$$

(see Theorem 4.1 for details) by checking for a nontrivial (passing through at least one nonconstant function symbol), positive-weight cycle starting from a variable node in the final graph, where the weights are determined by adding the cost of an arc traversed if $\text{dir} = \leftarrow$, and subtracting if $\text{dir} = \rightarrow$. (Arcs joining a function symbol with its arguments can be traversed, but have no effect on the cost.)

```plaintext
procedure Propagate (tnode, hnode, cost, dir);
begin
if $\{\text{tnode, hnode}\} \cap \mathcal{V} \neq \emptyset$ then
  Add-arc (tnode, hnode, cost, dir)
else
  if tnode $\neq$ hnode then fail
```


/* Failure occurs since there is disagreement between two nonvariable function symbols (i.e., a root-conflict occurs). */

else
    for j := 1 to n do
        Propagate (jth child of tnode, jth child of hnode, cost, dir);
end (* procedure Propagate *)

procedure Add-arc (tnode, hnode, cost, dir);
begin
    if tnode ∈ V and dir = → then
        add the arc (tnode, hnode, cost, dir) to the graph
    else if hnode ∈ V then
        add the arc (hnode, tnode, cost, reverse-dir) to the graph
    else
        /* tnode ∈ V and hnode is not */
        add the arc (tnode, hnode, cost, dir) to the graph;
        if adding an arc introduces multiple arcs from a tnode then
            if there is another arc from tnode to hnode (even marked for deletion) then
                call the procedure Multi-edge2 (tnode);
            else
                /* the hnodes of the multiple arcs are distinct */
                call Multi-edge1 (tnode);
        end (* procedure Add-arc *)

procedure Multi-edge1 (tnode);
begin
    /* Note that tnode must be a variable; assume that the two arcs emanating from tnode are (tnode, h1, c1, dir1) and (tnode, h2, c2, dir2). (The two arcs under consideration are the one just added and the one that is not marked for deletion (there will be exactly one).) Assume w.l.o.g. that c1 ≥ c2. While the algorithm runs, we may want to simply mark certain edges as deleted rather than actually deleting them, as it may be useful to have access to it later, when duplicate arcs are added between two nodes. */
     CASE 1: dir1 = dir2 = →
        mark for deletion the first arc
        /* Congruence will be preserved by the new edge(s) added. */
        call Propagate (h2, h1, c1 = c2, →);
     CASE 2: dir1 = dir2 = ←
        mark for deletion the first arc;
        /* Congruence will be preserved by the new edge(s) added. */
        call Propagate (h1, h2, c1 = c2, →);
        /* This case uses cancellativity of ρ cited in Theorem 4.4. */
     CASE 3: dir1 ≠ dir2
mark for deletion the first arc;
   /* Congruence will be preserved by the new edge(s) added. */
call Propagate (h2, h1, c1 + c2, →);
   /* Note that doubling of cost may occur with each call that matches CASE
   3. Thus the cost can be exponential even if the number of calls is not. In the
   next section we show that this will not affect the running time of the algorithm
   adversely. */
end (* Multi-edge1 *)

procedure Multi-edge2 (tnode);
begin

   /* Note that, as above, tnode must be a variable; assume that the two arcs
eemanating from tnode are (tnode, h1, c1, dir1) and (tnode, h1, c2, dir2).
Actually, there may be three arcs between the two nodes under consideration
(the new one, one undeleted, and one marked for deletion), but no more
than two with the same dir field, and these are the two to be considered in
the following should this occur. Assume w.l.o.g. that c1 ≥ c2, and that the
original arc may have been marked as deleted (see above). */

   if h1 = tnode, then
      if gcd(c1, c2) ≠ c2, then
         delete both arcs from the graph,
         replacing them with a new arc by calling Add-arc (tnode, tnode, gcd(c1, c2));
         /* Justification for using gcd(c1, c2) can be seen by examining the simplifi-
         cation procedure of the two corresponding rules by one another, which
         exactly follows the Euclidean greatest common divisor algorithm. */
      else gcd(c1, c2) = c2, delete the arc with cost c1 from the graph;
   else
      /* Assume tnode ≠ h1. */
      CASE 1: dir1 = dir2 = →
      call Propagate (h1, h1, c1 - c2, →);
      call Propagate (tnode, tnode, c1 - c2, →);
      delete the arc with cost c1;
      /* Furthermore, note that using the fact that \( p^{(c1-c2)} \) \( tnode = tnode \), we have
      that if c1 = (c1 - c2) + (c1 - c2) + \ldots + (c1 - c2) + r, and that if c2 ≥ c1/2
      then we can delete the arc with cost c2 and add the arc corresponding to
      \( p^{(c1-c2)} \) tnode = h1 by calling Propagate (tnode, h1, r, →). Thus we have */
      if c2 ≥ c1/2 then
         delete the arc with cost c2
         and call Propagate (node, h1, r, →);
         /* Note that if a new arc is added it has cost strictly less than c2/2, thus
         the maximum number of times that this can occur is bounded by \( \log_2 \)
         maximum-cost. This is discussed more fully in the next section. */
      CASE 2: dir1 = dir2 = ←
CASE 3: dir1 ≠ dir2

/* We will only consider this case when there is no other arc from tnode (even marked for deletion) which can match with CASE 1 or CASE 2 above. Thus there will be exactly two edges between tnode and h1, and their dir values will disagree. */
call Propagate (h1, h1, c1 + c2, \rightarrow);
call Propagate (tnode, tnode, c1 + c2, \rightarrow);
mark the arc with cost i for deletion;

/* Note that although the cost of calls to Propagate increases here, it can only happen once for each (tnode, h1) pair. Any subsequent calls on this pair of nodes will match with CASE 1 or CASE 2 above. */
end (Multi-edge2 *)

5.2. A polynomial bound

We provide a sketch of why Algorithm B is guaranteed to run in time polynomial in the size (call it \( n \)) of the terms.

The initial call to Propagate directly results in no more calls to Add-arc than there are occurrences of variables in the original terms. It is only when multiple arcs are added to the graph emanating from a single variable node that problems (a potentially exponential number of calls to Propagate) can occur. Note that costs on arcs increase in calls to Multi-edge1 whenever the dir arguments disagree (in fact, they may double when this occurs). This also happens when a self-loop is added in the third case of Multi-edge2. Despite the increase in cost mentioned above, we note that we can only add \( O(n^2) \) arcs to this graph before we begin to add duplicate arcs (with the same dir arguments) between nodes, thus guaranteeing that after a "short" time, calls are made to Multi-edge2, each of which either match with the "self-loop" case or with the cited CASE 1 or CASE 2. The key points are the following.

1. When there are multiple arcs emanating from a variable we can eliminate all but one (and possibly a self-loop on the variable node) by adding new arcs through calls to Propagate, without changing the congruence (this makes the final check for a nontrivial cycle considerably easier); one can easily check this claim by examining the rules that correspond to such arcs.

2. When an attempt is made to introduce a new arc between two nodes (not necessarily distinct) where some arc already exists, the cost either does not change (no call to Propagate results), or the cost argument to the new arc introduced between the two nodes is less than half the smaller of the costs of the two original arcs. (The key point here is that before this duplication of arcs happens the cost may become exponential but no larger. If this cost is cut in half each time a new arc is introduced, the algorithm must stabilize (terminate) in polynomial time.)

To understand why the second claim above holds, we examine what happens in CASE 1 of the procedure Multi-edge2 (the other cases are similar): Let the two arcs
be \((\text{tnode, h1, i, dir1})\) and \((\text{tnode, h1, j, dir2})\). We assume without loss of generality that \(i > j\). As mentioned in the comments to the procedure Multi-edge2, using the facts that \(p^{(i-j)}\text{tnode} = \text{tnode}\), \(i = (i - j)k + r\), and \(j = (i - j)(k - 1) + r\), with \(k \geq 1\), and \(r\) (the remainder) strictly less than \((i - j)\), we have the following:

- If \(j < i/2\), then \(r = j\) and no change occurs between the two nodes, since the arc to be added, \((\text{tnode, h1, j, dir})\), is already in the graph.

**Fig. 1.** The graph of the initial call on terms 
\[h(f(x, y), s_u), h(s_u, f(s_y, g(s_y)))\].

**Fig. 2.** After propagating the initial call.

**Fig. 3.** Multiple arcs from \(s_u\) are resolved.

**Fig. 4.** Again multiple arcs are introduced as a result of propagation.

**Fig. 5.** Finally, the graph stabilizes, with failure detected because of the nonnegative cost cycle from \(s_y\) to itself through \(g\).
• If $j \geq i/2$ then we can delete the arc with cost $i$, mark for deletion the arc with cost $j$ and add the arc corresponding to $p' = \text{node} = h1$. We maintain that the cost $r$ on the new arc is strictly less than $j/2$. This is true for the following reason:
• If $j \geq i/2$, then $(i-j)<i/2$, and $j = (i-j)k + r$, with $k \geq 1$. As a result, if $r \geq j/2$, since $k \geq 1$, $(i-j)<j/2$, thus $j = (i-j)k + (i-j) + r'$, contradicting the assumption that $r < (i-j)$.

Since the introduction of new arcs is (after a polynomial number of iterations at each of which the costs on arcs may as much as double) effectively limited by the number of bits needed to represent the largest cost, it is a straightforward conclusion that the procedure runs in polynomial time.

5.3. An example

We give an example above, illustrating failure of semi-unifiability and its detection using the method outlined above (in the interest of clarity we have included arcs between nonvariable terms although these would not actually be added by the algorithm): see Figs. 1–5.

References

Semi-unification

