Fast left-linear semi-unification

F. Henglein

RUU-CS-89-33
December 1989

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Fast left-linear semi-unification

F. Henglein

Technical Report RUU-CS-89-33
December 1989

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

Fast Left-Linear Semi-Unification

Fritz Henglein
Department of Computer Science
Utrecht University
PO Box 80.089
3508 TB Utrecht
The Netherlands
Internet: henglein@cs.ruu.nl

December 22, 1989

Abstract

Semi-unification is a generalization of both unification and matching with applications in proof theory, term rewriting systems, polymorphic type inference, and natural language processing. It is the problem of solving a set of term inequalities $M_1 \leq N_1, \ldots, M_k \leq N_k$, where $\leq$ is interpreted as the subsumption preordering on (first-order) terms. Whereas the general problem has recently been shown to be undecidable, several special cases are decidable.

Kfoury, Tiuryn, and Urzyczyn proved that *left-linear semi-unification (LLSU)* is decidable by giving an exponential time decision procedure. We improve their result as follows.

1. We present a generic polynomial-time algorithm L1 for LLSU, which shows that LLSU is in P.
2. We show that L1 can be implemented in time $O(n^3)$ by using a fast dynamic transitive closure algorithm.
3. We prove that LLSU is P-complete under *log*-space reductions, thus giving evidence that there are no fast (NC-class) parallel algorithms for LLSU.

As a corollary of the proof of P-completeness we obtain that LLSU with only 2 term inequalities is already P-complete.

We conjecture that L1 can be implemented in time $O(n^2)$, which is the best that is possible for the solution method described by L1. The basic question as to whether another solution method may admit even faster algorithms is open. We conjecture also that LLSU with 1 inequality is also P-complete.

1 Introduction

Semi-unification is a generalization of both unification and matching with applications in proof theory [Pud88], term rewriting systems [Pur87, KMNS89], polymorphic type inference [Hen88b, KTU89a, Lei89b], and natural language processing [DR89]. Because of its fundamental nature it may be expected to find even more applications.

Whereas general semi-unification was long believed to be decidable, Kfoury, Tiuryn and Urzyczyn recently gave an elegant reduction of the *boundedness* problem for deterministic Turing Machines to semi-unification [KTU89b]. By adapting a proof for a similar problem attributed to

Hooper [Hoo65] they showed that boundedness is undecidable, which implies the undecidability of semi-unification.

Several special cases of semi-unification have been shown to be decidable: uniform semi-unification (solving a single term inequality) [Hen88b, Pud88, KMNS89], semi-unification over two variables [Lei89a], and left-linear semi-unification [KTU89a]. Pudlak showed that general semi-unification can be effectively reduced to semi-unification over two inequalities [Pud88]; thus bounding the number of inequalities by a number greater than one does not simplify the problem. In drastic contrast, Kapur, Musser, Narendran and Stillman gave a polynomial time procedure for uniform semi-unifiability (single inequality) [KMNS89].

In this article we investigate the special case of *left-linear semi-unification (LLSU)*. This problem was first addressed by Kfoury, Tiuryn, and Urzyczyn [KTU89a]. They were able to show that left-linear semi-unification is decidable, and a closer look at their decision procedure reveals that left-linear semi-unification is in DEXPTIME. We improve this result by showing that left-linear semi-unification is polynomial time decidable. The present a generic algorithm, Algorithm L1, for LLSU that is implementable in polynomial time. An implementation based on a fast dynamic transitive closure algorithm [LPvL87, Yel88] (with only edge additions) yields an $O(n^3)$ time LLSU procedure where n is the size of the given (left-linear) semi-unification problem. Dynamic transitive closure seems too general and powerful a method for LLSU, and we conjecture that L1 can be improved to run in time $O(n^2)$. This is best possible for any method based on L1 since as many as n^2 edges are added to an initially sparse graph on n nodes in L1. The question as to whether there is a linear-time algorithm for left-linear semi-unification or, in fact, any algorithm asymptotically faster than $O(n^2)$ is left open.

We also show that, even though left-linearity is a very strong condition on input instances, it still is not strong enough to admit fast (NC-class) parallel algorithms unless NC = P. Specifically, we show that LLSU is P-complete under *log*-space reductions by adapting a well-known proof of Dwork, Kanellakis, and Mitchell for showing the P-completeness of unification [DKM84].

The outline of the rest of the paper is as follows. In section 2 we define general and left-linear semi-unification, and we present a general semi-unification algorithm, algorithm A [Hen89]. Observing the behavior of algorithm A on left-linear problem instances yields the critical insight that permits "speeding up" A to run in polynomial time. The result is Algorithm L1. In section 3 we present our polynomial time algorithm L1 for LLSU over the alphabet $\mathcal{A}_2$, which consists of a single binary function symbol, and show that a dynamic transitive closure based implementation has time complexity $O(n^3)$. In section 4 we show that left-linear semi-unification is P-complete. We conclude with some final remarks and open problems in section 5.

2 General semi-unification

In this section we present definition and properties of semi-unification and an (semi-)algorithm, Algorithm A, for solving general semi-unification problem instances. Most of this is a rehash from a previous paper [Hen89], but it is very instructive in giving insight into the correctness of our polynomial time algorithm for LLSU.

2.1 Definition and properties of semi-unification

A *ranked alphabet* $\mathcal{A} = (F, a)$ is a finite set F of *function symbols* together with an *arity* function a that maps every element in F to a natural number (including zero). A function symbol with arity 0 is also called a *constant*. The set of *variables* V is a denumerable infinite set disjoint from F. The *terms* over A and V is the set $T(\mathcal{A}, V)$ consisting of all strings generated by the grammar

$$M ::= x|c|f(\underbrace{M,\ldots,M}_{k\text{times}})$$

where f is a function symbol from $\mathcal{A}$ with arity $k > 0$, c is a constant, and x is any variable from V. Two terms M and N are equal, written as $M = N$, if and only if they are identical as strings.

A *substitution* σ is a mapping from V to $T(\mathcal{A}, V)$ that is the identity on all but a finite subset of V. The set of variables on which σ is *not* the identity is the *domain* of σ. Every substitution $\sigma : V \to T(\mathcal{A}, V)$ can be naturally extended to $\sigma : T(\mathcal{A}, V) \to T(\mathcal{A}, V)$ by defining

$$\sigma(f(M_1,\ldots,M_k)) = f(\sigma(M_1),\ldots,\sigma(M_k)).$$

A term M *subsumes* N (or N *matches* M), written $M \leq N$, if there is a substitution ρ such that $\rho(M) = N$. If N matches M then there is exactly one such ρ as required in the above definition whose domain is contained in the set of variables occurring in M. We call it the *quotient* substitution of M and N and denote it by N/M.

Given a set of pairs of terms $S = \{(M_1, N_1),\ldots,(M_k, N_k)\}$ a substitution σ is a *semi-unifier* of S if $\sigma(M_1) \leq \sigma(N_1),\ldots,\sigma(M_k) \leq \sigma(N_k)$. S is *semi-unifiable* if it has a semi-unifier. Semi-unifiability is reminiscent of both unification (because of σ being applied to both the left- and right-hand components of S) and matching (because the resultant right-hand sides have to match their corresponding right-hand sides), but it is in fact much more general than both, evidenced by the recent undecidability result for semi-unifiability.

In the context of semi-unification we shall call a set of pairs of terms a *system of inequalities* and write it also

$$\begin{array}{rcl} M_1 & \stackrel{?}{\leq} & N_1 \\ M_2 & \stackrel{?}{\leq} & N_2 \\ & \cdots & \\ M_k & \stackrel{?}{\leq} & N_k \end{array}$$

The question mark over the inequality symbol is to indicate that these are not *valid* inequalities, but are supposed to be *solved*; that is, a substitution is to be found that makes them valid.

As shown in [Hen88a], every semi-unifiable system of inequalities has a most general semi-unifier if the notion of generality on substitutions is properly defined.

Example:

Let $\mathcal{A}_2$ be a ranked alphabet with only one function symbol f, which has arity 2; $V = \{x_1, x_2,\ldots,x_i,\ldots\}$.

Consider the semi-unifiable system of inequalities S_0:

$$\begin{array}{rcl} f(f(x_1,x_2),x_3) & \stackrel{?}{\leq} & x_4 \\ x_4 & \stackrel{?}{\leq} & f(x_3,f(x_2,x_5)) \end{array}$$

Possible semi-unifiers are $\sigma_0 = \{x_3 \mapsto f(x_6,x_7), x_4 \mapsto f(f(x_8,x_9),f(x_{10},x_{11}))\}$ and $\sigma_1 = \{x_3 \mapsto f(x_2,x_2), x_4 \mapsto f(f(x_2,x_2),f(x_2,x_2)), x_5 \mapsto x_2\}$ (σ_0 is a most general semi-unifier of S_0) since, after substituting σ_0, respectively σ_1, in S_0, we get the valid inequalities

$$\begin{array}{rcl} f(f(x_1,x_2),f(x_6,x_7)) & \leq & f(f(x_8,x_9),f(x_{10},x_{11})) \\ f(f(x_8,x_9),f(x_{10},x_{11})) & \leq & f(f(x_6,x_7),f(x_2,x_5)), \end{array}$$

respectively

$$\begin{aligned} f(f(x_1,x_2),f(x_2,x_2)) &\leq f(f(x_2,x_2),f(x_2,x_2)) \\ f(f(x_2,x_2),f(x_2,x_2)) &\leq f(f(x_2,x_2),f(x_2,x_2)). \end{aligned}$$

The quotient substitutions for these inequalities are $\rho_0^1 = \{x_1 \mapsto x_8, x_2 \mapsto x_9, x_6 \mapsto x_{10}, x_7 \mapsto x_{11}\}$ and $\rho_0^2 = \{x_8 \mapsto x_6, x_9 \mapsto x_7, x_{10} \mapsto x_2, x_{11} \mapsto x_5\}$, respectively $\rho_1^1 = \{x_1 \mapsto x_2\}$ and $\rho_1^2 = \{\}$.

End of example

2.2 Algorithm A

Algorithm A is a general (semi-)algorithm for computing the most general semi-unifier of a system of inequalities. Even though it is bound not to terminate for some non-semi-unifiable inputs it catches many non-semi-unifiable systems of inequalities due to an *extended occurs check*, which is a generalization of the conventional occurs check in unification algorithms.

The algorithm operates on a graph-theoretic representation of systems of inequalities, both to achieve practically acceptable performance and to aid in the analysis of some combinatorial properties. Since intermediate steps of the algorithm can introduce *equations* $M \stackrel{?}{=} N$ between terms, not just inequalities, the representation, called an *arrow graph*, in fact encodes systems of equations and inequalities. Because a formal description of arrow graphs is notoriously cumbersome, we give a brief, but hopefully clear, informal definition below.

A *term graph* is an acyclic graph that represents sets of terms over a given alphabet $\mathcal{A} = (F, a)$ and set of variables V. It consists of a set of nodes, N, a subset of which is labeled with function symbols from F, and the rest of which is labeled with variables from V. If f is a function symbol with arity k, $k \geq 0$, every node n labeled with f has exactly k *ordered children*; i.e., there are k directed *term edges* originating in n and labeled with the numbers 1 through k. The variable labeled nodes have no children, and for every variable x there is at most one node labeled with x. The nodes together with the tree edges form a normal directed graph, and if it is acyclic, then we say the term graph is *acyclic*.

Every node in an acyclic term graph can be interpreted as a term; for example, if n is a node labeled with function symbol f, and its children are $n_1, \ldots, n_k$ (in this order) representing terms $M_1, \ldots, M_k$, then n represents the term $f(M_1, \ldots, M_k)$. Note that for every set of terms there is an easily constructed, but generally non-unique term graph such that every term is represented in it.

Example:

The term graph in Figure 1 represents the terms $f(f(x_1, x_2), x_3), x4$ and $f(x_3, f(x_2, x_5))$ occurring in the system of inequalities S_0 (see previous example).

End of example

A term graph can represent all the terms occurring in a system of equations and inequalities. An *arrow graph* is a term graph with two additional *kinds* of edges: *Equivalence edges* encode equations, and *arrows* encode inequalities.

Equivalence edges are represent an equivalence relation on the nodes of the arrow graph. They can be thought of as *undirected edges*, This is the notion we shall adopt, but we shall always assume that for every (undirected) path from node n_1 to node n_2 via equivalence edges (only) there is also an equivalence edge directly between n_1 and n_2. If there is an equivalence edge between n_1 and n_2 we write $n_1 \sim n_2$.

Arrows are directed edges labeled by natural numbers, which indicate to which inequality in a given system of equations and inequalities an arrow corresponds. We call the labels of arrows *colors*, and we write $n_1 \stackrel{i}{\rightarrow} n_2$ if there is an i-colored arrow pointing from n_1 to n_2.

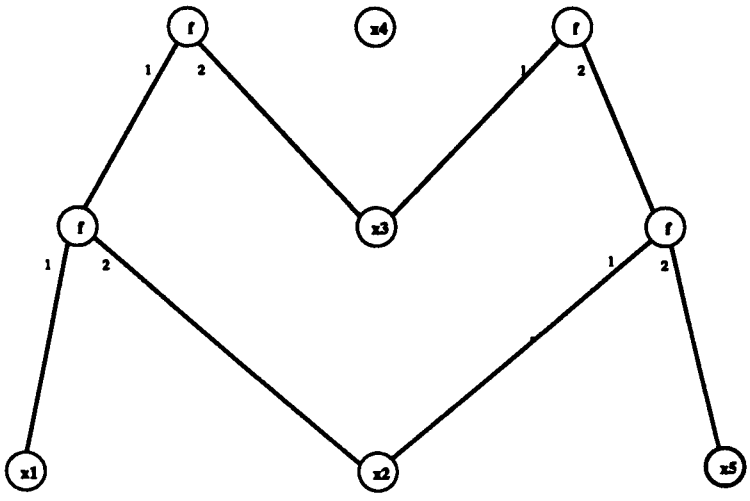

Figure 1: A term graph representation of a set of terms

To summarize, an arrow graph is a term graph with additional edges: undirected equivalence edges and directed arrows. Note that an arrow graph has three different kinds of edges: term edges, equivalence edges, and arrows.

An *arrow graph representation* of a system of inequalities S,

$$\begin{array}{lcl} M_1 & \stackrel{?}{\leq} & N_1 \\ M_2 & \stackrel{?}{\leq} & N_2 \\ & \cdots & \\ M_k & \stackrel{?}{\leq} & N_k, \end{array}$$

is a term graph G with (not necessarily distinct) nodes $m_1, \ldots, m_k, n_1, \ldots, n_k$ representing the terms in S, and arrows from m_i (representing M_i) to n_i (representing N_i) for $1 \leq i \leq k$ that have pairwise distinct labels. There are no equivalence edges. (In other formulations systems of equations and inequalities are input instances for semi-unification, in which case equivalence edges are used to represent equations in the arrow graph representation.)

Example:

An arrow graph representation of system of inequalities S_0 is in Figure 2. Term edges are shown in straight, bold lines; arrows in curved, thin lines. There are no equivalence edges.

End of example

Algorithm A operates on arrow graphs. It repeatedly rewrites the arrow graph representation of a system of inequalities S by nondeterministically "applying" some closure rules until no rule can be applied any more. At that point it indicates whether S is semi-unifiable and, if so, outputs a most general semi-unifier of S. The algorithm is described in detail in Figure 3 for alphabet $\mathcal{A}_2$[1], and the closure rules are also depicted graphically in Figure 4.

Example:

[1] $\mathcal{A}_2$ is the alphabet consisting of a single binary function symbol f. Semi-unifiability over any alphabet is *log*-space reducible to semi-unifiability over $\mathcal{A}_2$ [Hen89]. This reduction does not, however, preserve left-linearity.

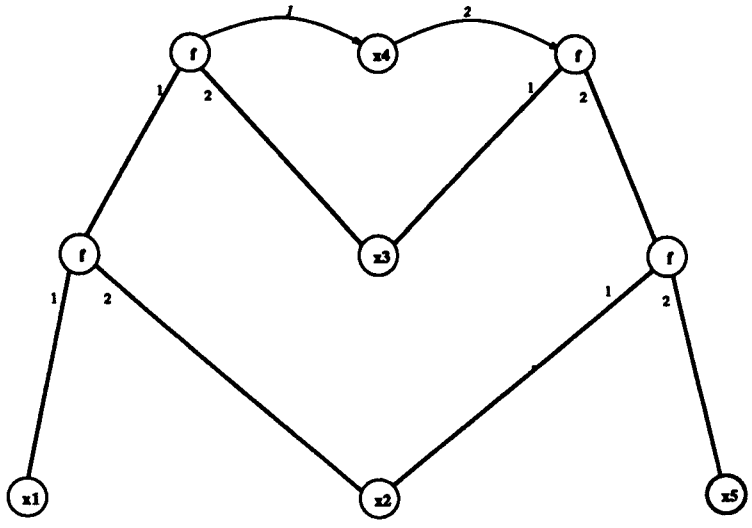

Figure 2: An arrow graph representation of a system of inequalities

The effect of Algorithm A on the arrow representation of the system of inequalities S_0 is shown in Figures 5 and 6. In the initial arrow graph representation (see Figure 2) only rule 4b in Algorithm A (see Figure 3) is applicable. The arrow graph after after its application is shown in Figure 5. Immediately afterwards rule 3a can be applied to add an arrow (colored with 1) pointing to the new f-labeled node, and application of rule 2 adds 1-colored arrows pointing to the new variable-labeled nodes, at which point rule 4b is applicable again. The algorithm finally terminates with the arrow graph shown in Figure 6. Since this final arrow graph is not $\square$ the given system of inequalities is semi-unifiable, and a most general semi-unifier can be read off from the final arrow graph: $\{x_3 \mapsto f(x_9, x_{12}), x_4 \mapsto f(f(x_8, x_{11}), f(x_{10}, x_{13}))\}$, which is equivalent to σ_0 modulo renaming of variables.

End of example

2.3 Left-linear semi-unification

A term is *linear* if every variable has at most one occurrence in it. For example, $f(x_1, x_2)$ is linear (assuming $x_1 \neq x_2$), but $f(x_1, x_1)$ is not. A system of inequalities S is *left-linear* if every left-hand side term in S is linear. *Left-linear semi-unification* is the problem of computing most general semi-unifiers of left-linear systems of inequalities, and *left-linear semi-unifiability (LLSU)* is the problem of deciding whether a given left-linear system of inequalities is semi-unifiable.

Example:

Recall the system of inequalities S_0,

$$\begin{aligned} f(f(x_1, x_2), x_3) &\stackrel{?}{\leq} x_4 \\ x_4 &\stackrel{?}{\leq} f(x_3, f(x_2, x_5)). \end{aligned}$$

The left-hand sides are the terms $f(f(x_1, x_2), x_3)$ and x_4. Since they are linear, S_0 is an instance of left-linear semi-unification (over alphabet $\mathcal{A}_2$).

Let G be an arrow graph. Apply the following rules (depicted also in Figure 4) until convergence:

1. If there exist nodes m and n labeled with f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \sim n$ then add $m_1 \sim n_1$ and $m_2 \sim n_2$.
2. If there exist nodes m and n labeled with f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \xrightarrow{i} n$ then add arrows $m_1 \xrightarrow{i} n_1$ and $m_2 \xrightarrow{i} n_2$.
3. If there exist nodes m_1, m_2, n_1, and n_2 such that
   (a) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $n_1 \xrightarrow{i} n_2$ then add $m_2 \sim n_2$;
   (b) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $m_2 \sim n_2$ then add an arrow $n_1 \xrightarrow{i} n_2$.
4. (a) (Extended occurs check) If there is a path consisting of arrows of any color (arrow path) from n_1 to n_2 and n_2 is a proper descendant of n_1, then reduce to the improper arrow graph $\square$.[a]
   (b) If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with f and has children m_1, m_2, n is labeled with a variable, and $n \sim n''$ implies that n'' is variable labeled, and there is an arrow $m \xrightarrow{i} n$ then create new nodes n', n'_1, n'_2 and label n' with f, label n'_1 and n'_2 with new variables x' and x'', respectively; make n'_1, n'_2 the children of n'; and add $n \sim n'$.

[a] Node n' is a *descendant* of node n if it there is a path from n to n' consisting of term edges (traversed in forward direction) and equivalence edges (traversed in any direction); n' is a *proper* descendant if there is a path with at least one term edge.

Figure 3: Algorithm A

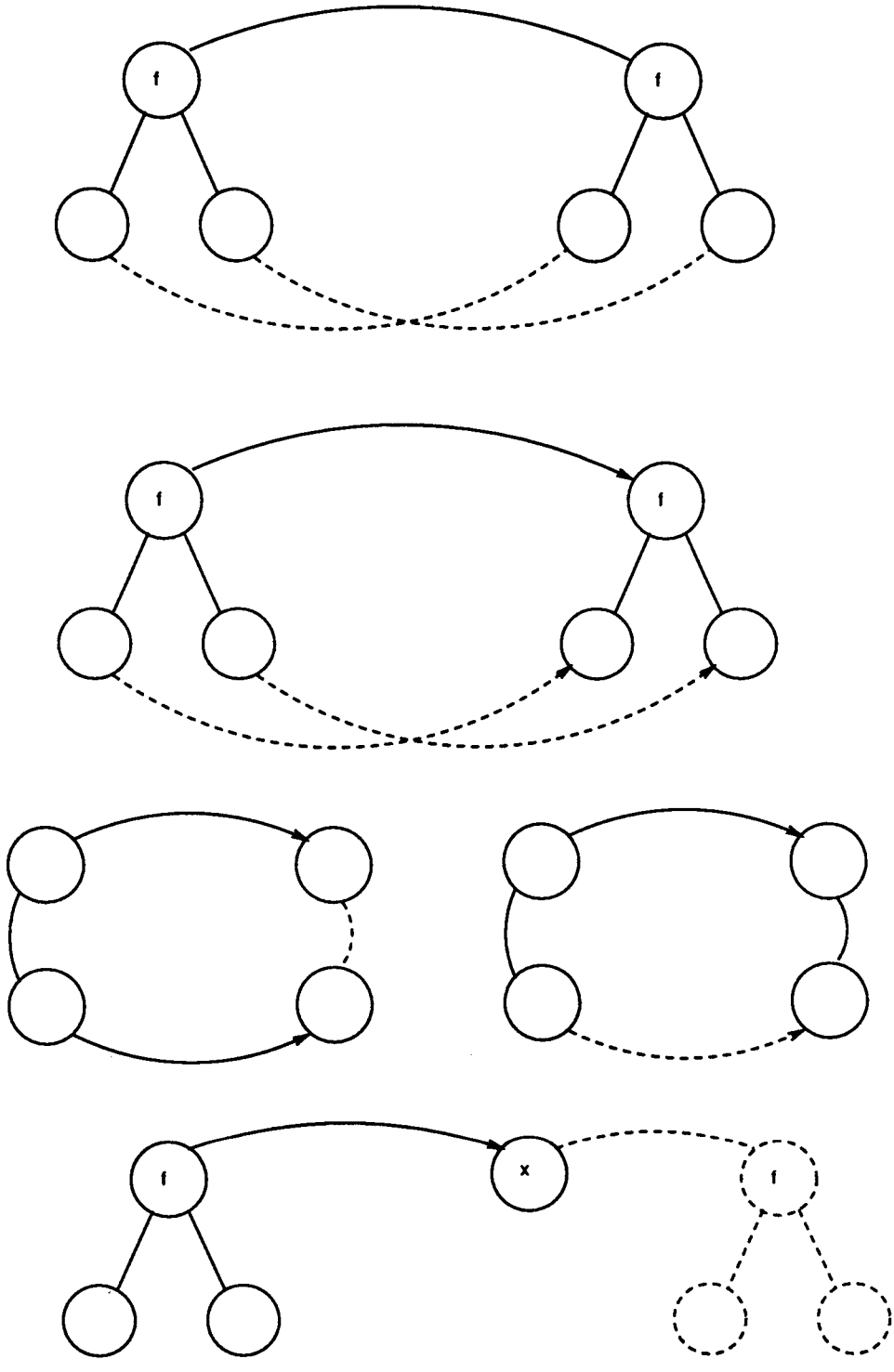

Figure 4: Closure rules

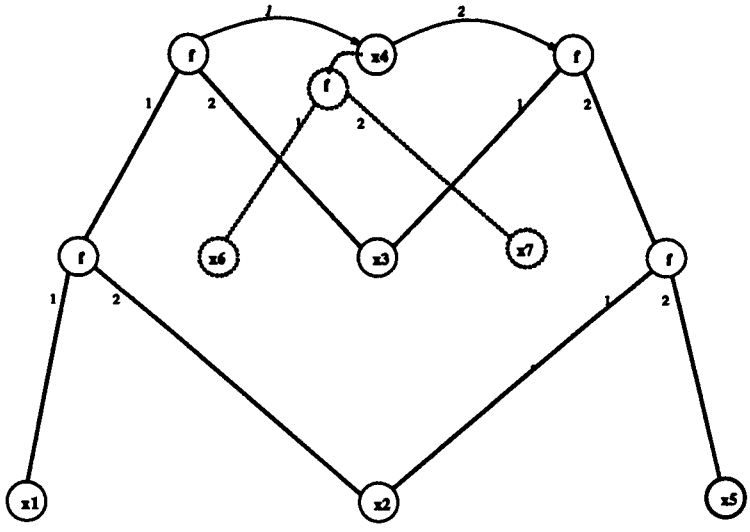

Figure 5: After application of a closure rule

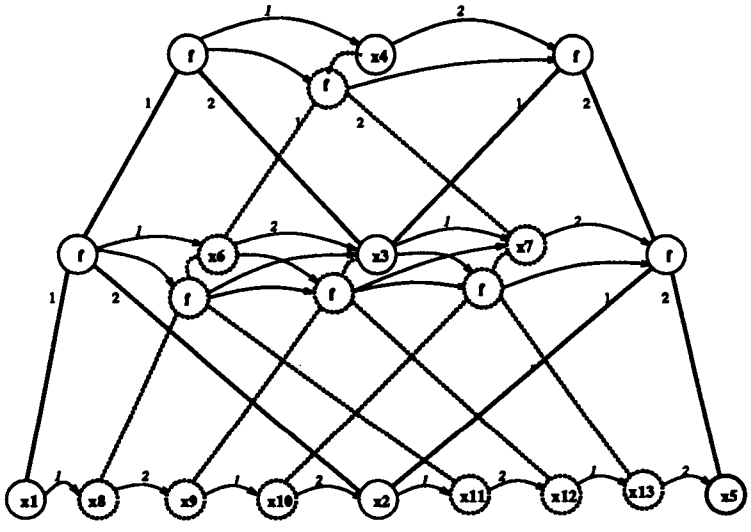

Figure 6: A final arrow graph

(4b)

If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with f and has children m_1, m_2, n is labeled with a variable, and $n \sim n'$ implies that n' is variable labeled, and there is an arrow $m \xrightarrow{i} n$ then create new nodes n'_1, n'_2 and change the label of n to be f, label n'_1 and n'_2 with new variables x' and x'', respectively; and make n'_1, n'_2 the children of n.

Figure 7: Modified rule for expanding arrow graph

End of example

For reasons that will become clear below we can modify rule 4b in an almost trivial fashion. If rule 4b is applicable because there is an arrow from an f-labeled node m to a variable labeled node n, it prescribes that a new f-labeled node n' with new variable labeled children n'_1, n'_2 and an equivalence edge $n \sim n'$ be added to the arrow graph. The *modified* rule 4b is as follows. Instead of adding three new nodes n', n'_1, n'_2 and an equivalence edge $n \sim n'$ we add only two new nodes, n'_1, n'_2 (labeled with new variables), *change* the label of n from a variable to f, and make n'_1, n'_2 the children of n. (The old label of n is remembered in some auxiliary data structure. This information is only needed for reading off the most general semi-unifier from a final arrow graph, but not for the execution of Algorithm A itself.) This modification is given in detail in Figure 7. Henceforth, we shall assume that Algorithm A uses the modified rule 4b.

The correctness of Algorithm A is proved in great detail in [Hen89], and it is a minor issue to verify that the modified version of rule 4b preserves correctness.

Example:

The final arrow graph after executing modified Algorithm A on the arrow graph representation of system S_0 of Figure 2 is shown in Figure 8. Note that the arrow graph contains no equivalence edges, and only rules 2 and 4b of Algorithm A were applied.

End of example

Let us consider an arrow graph representation of a left-linear system of inequalities S and one of its nodes m that represents a left-hand side. Since S is left-linear the subgraph rooted at m is a tree; that is, there is one and only one path consisting of term edges from m to any of the nodes reachable from m via term edges. Executing Algorithm A on the arrow graph of Figure 2 we see that at no point rule 3a, the only rule that could possibly introduce the first equivalence edge, is applicable. This is not just a peculiar property of the specific example, but holds for every (arrow representation of a) left-linear system of inequalities.

An *execution (of Algorithm A)* is a sequence of arrow graphs $(G_0, \ldots, G_i, \ldots)$ in which every component is derived from its predecessor by application of one of the closure rules of Algorithm A (see Figure 3 or Figure 4).

Theorem 1 *Let $(G_0, \ldots, G_i, \ldots)$ be an execution of Algorithm A. If G_0 is an arrow graph representation of a left-linear system of inequalities, then G_{i+1} is derived from G_i by rule 2 or rule 4.*

Proof:

For $(G_0, \ldots, G_i, \ldots)$ we can prove by induction on i that the following properties hold for every $G_i \neq \Box$:

1. Every node has at most one outarrow of any given color.

2. The subgraph rooted at any node with an outarrow (of any color) is a tree

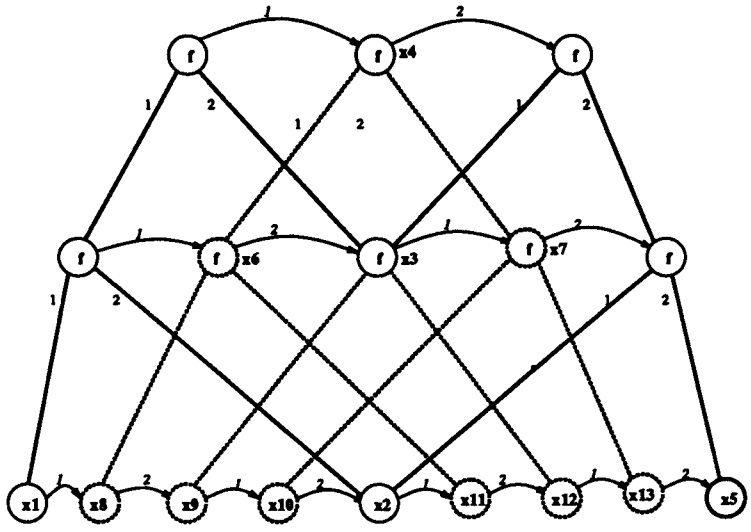

Figure 8: A final arrow graph produced by modified Algorithm A

This implies the theorem since it guarantees that neither rule 3 nor rule 1 is applicable to any of the G_i's.

End of proof

This theorem yields the critical insight for speeding up Algorithm A for the special case of solving left-linear semi-unification problems. We will give very informal considerations below that will lead us, directly from the observation of theorem 1, to the polynomial time LLSU-algorithm L1 below. L1 is presented in and its correctness is proved in the following section.

A quite immediate simplification of Algorithm A is that, for left-linear semi-unification, rules 3 and 1 are not needed. But further simplifications are possible. Note that the color maintained with every arrow is only needed as a criterion for applying rule 3. For example, if a node n has two outarrows with *equal* color to distinct nodes n' and n'' an equivalence edge $n' \sim n''$ has to be added, but if the two arrows have *different* colors no such equivalence edge has to be added. Because, by theorem 1, the first case can never happen for left-linear systems of inequalities, we can dispense with the coloring information on arrows altogether.

But without color information we may also maintain the arrows transitively closed; that is, we can maintain an arrow from n_1 to n_2 (all arrows are uncolored now) with every arrow *path* from n_1 to n_2. This is not an advantage by itself, but it does away with the need to apply rule 4b at all, if we adopt a modified extended occurs check rule (rule 4a). Basically, the only relevant effect of rule 4b on left-linear arrow graph representations is to create establish arrow paths between the children of f-labeled nodes that are in turn connected with an arrow path. This can be achieved directly, without adding new nodes, by maintaining the transitive closure of the arrows as indicated above or by generalizing rule 2 to apply arrow paths instead of only individual arrows. If there is an arrow path from a function symbol labeled node n_1 to another function symbol labeled node n_2 in the original algorithm, our strategy of maintaining an arrow

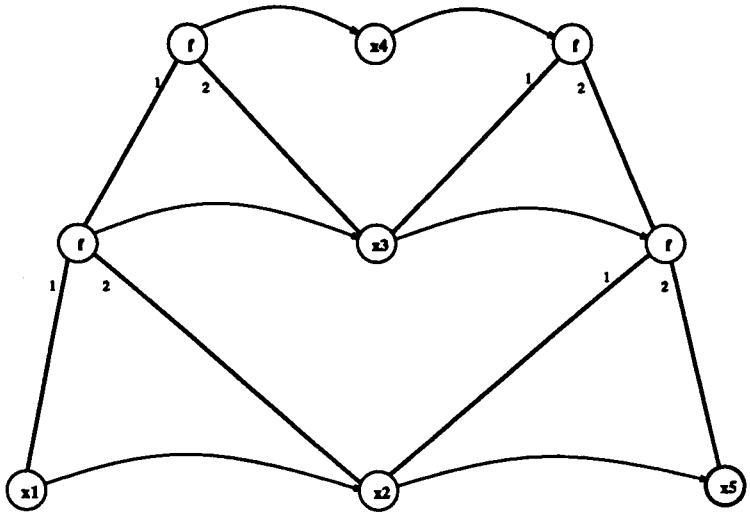

Figure 9: A final arrow with "shortcutting"

for every arrow path guarantees that there is also an arrow from n_1 to n_2, and applying rule 2, which propagates arrows downwards to the children of n_1 and n_2, will guarantee that there is an arrow from any child of n_1 to the corresponding child of n_2. We call the "contraction" of arrow paths to single arrows *shortcutting* since repeated copying with rule 4b is not necessary any more — it is "shortcut".

Example:

Consider again the system of inequalities S_0. The algorithm outlined informally above will simply propagate an arrow *path* between two f-labeled nodes to *single* arrows between their corresponding children with rule 2. Instead of the arrow graph of Figure 8 the we obtain the final arrow graph in Figure 9.

End of example

This leads to a polynomial time algorithm as can be easily seen from the fact that it is not necessary any more to add new nodes and all computational steps — maintaining the transitive closure and applying rule 2 — can be performed in polynomial time.

3 A polynomial time LLSU algorithm

In this section we present a polynomial time algorithm, Algorithm L1, for solving left-linear systems of inequalities over alphabet $\mathcal{A}_2$. In fact, the algorithm can be used to compute a most general semi-unifier, but we shall leave this aspect unexplored at this point. We show that L1 can be made to run in time $O(n^3)$ by adapting the fast dynamic transitive closure algorithm of La Poutré and van Leeuwen [LPvL87].

A *reduced arrow graph* is an arrow graph without equivalence edges and with only one color on arrows, which is consequently completely redundant and may be left off. We write $n \rightarrow n$ for an uncolored arrow from n to n'. A reduced arrow graph representation of a system of inequalities is simply an arrow graph representation with the colors on arrows left off.

Our first algorithm, Algorithm L1, is described in Figure 10. Roughly, it starts with a reduced

algorithm $L1(A$: **reduced arrow graph)**
while there exist nodes m_1, n_1 and m_2, n_2 in A and number i such that
n_1 is the i-th child of m_1 and n_2 is the i-th child of m_2,
m_1, m_2 are f-labeled (have the same function symbol label),
there is an arrow path from m_1 to m_2,
and there is no arrow from n_1 to n_2
do
add an arrow (arrow path) from n_1 to n_2 to A
end while;
if there is a cycle in A consisting arrows traversed in *forward* direction
and term edges traversed in *backward* direction (i.e., from child to parent)
then signal non-semi-unifiability
else signal semi-unifiability
end if;

Figure 10: Algorithm L1

arrow graph representation of a given system of inequalities, in which the colors of arrows are ignored. It then looks for arrow paths between f-labeled nodes and adds (uncolored) arrows between their corresponding children until this entails no more changes to the reduced arrow graph. In the final stage, the resultant reduced arrow graph is checked whether it contains a cycle that is made up of arrows — traversed in forward direction — and term edges — traversed from child to parent — and contains at least one term edge. If there is such a cycle we say the final arrow graph *fails* the *acyclicity test* and the algorithm signals non-semi-unifiability; otherwise, the final arrow graph *passes* the acyclicity test, and the algorithm signals semi-unifiability.

We first establish the correctness of L1 and then show that it can be implemented in polynomial time. To address correctness we need to recall some results on the structure of terms with respect to subsumption [Hue80].

Term subsumption is a preordering. The equivalence relation canonically induced by $\leq$ is defined by $M \cong N \Leftrightarrow M \leq N \wedge N \leq M$. We write $[M]$ for the $\cong$-equivalence class of term M; $T(\mathcal{A}, V)/_{\cong}$ for the set of equivalence classes of all terms over A and V; and $\leq$ for the partial order on $T(\mathcal{A}, V)/_{\cong}$ induced by the preorder $\leq$.

Theorem 2 $(T^{\Omega}/_{\cong}, \leq)$ *is a complete lattice.*

Proof: See [Hue80]. **End of proof**

In particular, every set of terms M has a least upper bound $\bigvee \mathcal{M}$ w.r.t. the subsumption preordering $\leq$ that is unique modulo $\cong$. The equivalence relation $\cong$ is also sometimes informally referred to as "renaming of variables". Note that every finite set of terms has a least upper bound whose variables are disjoint from the variables occurring in *any* fixed finite set.

Theorem 3 *Let A be a reduced arrow graph representing a left-linear system of inequalities S. If A signals non-semi-unifiability then S is not semi-unifiable. If A signals semi-unifiability then S is is semi-unifiable.*

Proof:

We shall show that if L1 fails the acyclicity test then S has no semi-unifier, and if L1 passes the acyclicity test then it has a semi-unifier. In fact the most-general semi-unifier can be extracted from the final, acyclic reduced arrow graph.

1. (L1 fails the acyclicity test)

   Assume S has a semi-unifier σ. Then every node n in the final arrow graph represents a unique term $T(n)$ with respect to σ according to the following rule.

$$T_\sigma(n) = \begin{cases} f(T_\sigma(n_1), T_\sigma(n_2)) & \text{if } n \text{ is } f\text{-labeled with children } n_1, n_2 \\ \sigma(x) & \text{if } n \text{ is labeled with variable } x \end{cases}$$

   We shall write $|M|$ for the size of M, measured in terms of the number of function symbol occurrences and variable occurrences in M. If σ is a semi-unifier of S, then it is easy to see that the final reduced arrow graph must satisfy the following relations. If $n \rightarrow n'$ then $|T_\sigma(n)| \leq |T_\sigma(n')|$, and if n is a child of n' then $|T_\sigma(n)| < |T_\sigma(n')|$. But if the final reduced arrow graph contains a cycle with at least one term edge then there is a node n with $|T_\sigma(n)| < |T_\sigma(n)|$, which is impossible. Consequently, S does not have a semi-unifier.

2. (L1 passes the acyclicity test)

   If L1 passes the acyclicity test, the final reduced arrow graph A' is acyclic in the sense that all cycles consist of arrows only. We define a term interpretation T for every node in A' as follows. Consider the strong components $C_1, \ldots, C_k$ of A' in topological order. For every node n in a strong component C_i we associate a term $T(n)$.

   Let $T(n)$ be defined for all nodes in components $C_1, \ldots, C_{i-1}$. Consider the predecessors $n_1, \ldots, n_k$ of node n in component C_i where, for all $j \in \{1, \ldots, k\}$, $n_j \in C_{j'}$ for some $j' < i$, and n is labeled with variable x. If $k = 0$ then $T(n) = x$. Otherwise, define the *preliminary* term interpretation $T_{\mathrm{prelim}}(n) = \bigvee\{T(n_1), \ldots, T(n_k)\}$. If n is f-labeled with children n_1, n_2 we define $T_{\mathrm{prelim}}(n) = f(T(n_1), T(n_2))$.

   Now, if C_i consists of the nodes $n, n', \ldots, n^{(l)}$, let

   $T = \bigvee\{T_{\mathrm{prelim}}(n), T_{\mathrm{prelim}}(n'), \ldots, T_{\mathrm{prelim}}(n^{(l)})\}$,

   and let $T', \ldots, T^{(l)}$ be $\cong$-equivalent to T with pairwise disjoint variables. Finally, define $T(n) = T, T(n') = T', \ldots, T(n^{(l)}) = T^{(l)}$.

   Note that T is not uniquely defined, but, most importantly, it has the property that for every pair of variable labeled nodes n, n' the sets of variables in $T(n), T(n')$ are disjoint.[2]

   Define substitution $\sigma = \bigcup\{x \mapsto T(n) : n \text{ is labeled with variable } x\}$. By construction of T we have that for every arrow $n \rightarrow n'$ in A', where n' is variable labeled, there is a substitution ρ such that $\rho(T(n)) = T(n')$; in particular, we can take $\rho = T(n')/T(n)$. By induction on the term structure in A' it follows that this holds also for an f-labeled node n'. Since it holds, in particular, for all arrows in the original arrow graph A this shows that σ is a semi-unifier. Some more analysis shows that it is a most general semi-unifier.

End of proof

Theorem 4 *Algorithm L1 is implementable in polynomial time.*

Proof:

Algorithm L1 can add at most n^2 new arrows to an arrow graph of size n, and the applicability of a single arrow addition step takes time at most $O(n^3)$ (with a naive transitive closure algorithm). Finally, the acyclicity testing step can be performed in time $O(n^2)$.

End of proof

[2] We assume that every least upper bound of a finite set of terms defined above has variables disjoint from all variables occurring in all previously defined term interpretations.

Corollary 1 *Left-linear semi-unifiability is in P.*

In the following theorem we show that adapting a fast dynamic transitive closure algorithm can be used to implement Algorithm L1 in time $O(n^3)$.

Theorem 5 *There is a dynamic transitive closure based implementation of Algorithm L1 that runs in $O(n^3)$ time for a left-linear system of inequalities of size n.*

Proof: (Sketch)

Since there can be as many as $O(n^2)$ additions of arrows in Algorithm A, a naive implementation that maintains the transitive closure of all arrows in $O(n^2)$ with respect to a single edge addition takes a total of $O(n^4)$ time. The algorithms of La Poutré/Van Leeuwen [LPvL87] and Yellin [Yel88], however, have accumulative cost of $O(n^3)$ and can be modified to permit finding a pair of f-labeled nodes n, n' whose children are not yet connected via arrows as a by-product of maintaining the transitive closure of the arrow graph. Consequently the total cost of the **while**-loop is $O(n^3)$. The final acyclicity test can be implemented in time $O(n^2)$ with a fast maximal strong components algorithm.

We give a sketch, due to Han La Poutré, of a modified dynamic transitive closure implementation that permits fast execution of the Boolean test in the while-loop of Algorithm L1 (see Figure 10) as part of updates after edge additions. Basically, we work with 5 copies, $V, V_{lc}, V_{rc}, V_{lp}, V_{rp}$, of the original nodes in the arrow graph A. Initially, we add edges as follows.

- If n_l is a left child of n in A then we add an edge from the copy of n_l in V_{lc} to the copy of n in V and an edge from the copy of n in V to the copy of n_l in V_{lp}.
- If n_r is a right child of n in A then we add an edge from the copy of n_r in V_{rc} to the copy of n in V and an edge from the copy of n in V to the copy of n_r in V_{rp}.
- If there is an arrow $n \rightarrow n'$ then we add an edge from the copy of n in V to the copy of n' in V.

Let us call the resulting directed graph G. We maintain the transitive closure of G in a bit matrix (of size $O(n^2)$). The while-loop in Algorithm L1 is executed as follows. Let S be the workset of pairs of nodes n, n' where n is a node in V_{lp} and n' in V_{lc} (or n in V_{rp} and n' in V_{rc}) and n reaches n' in G. These pairs represent the candidates for which an arrow has to be added. While there is a pair (n, n') in S, we delete it from S and, if there is no edge between the copies of n and n' in V (!) we add it, calculate the transitive closure of G, and update S accordingly; i.e., if a new node n' in V_{lc} (V_{rc}) becomes reachable from a node n in V_{lp} (V_{rp}), we add the pair (n, n') to S.

This sketch shows that finding update candidates can be performed as part of a dynamic transitive closure algorithm without additional asymptotic cost. Consequently, using the fast dynamic transitive closure algorithm for edge additions of La Poutré and van Leeuwen [LPvL87] yields an $O(n^3)$ implementation of Algorithm L1.

End of proof

Since the arrow additions are predetermined by Algorithm L1 itself, we believe that there is an $O(n^2)$ implementation of L1 based on a dynamic depth-first search algorithm. Since there can be as many as $O(n^2)$ arrows added by L1, this would be the best that is possible for L1. Of course, other algorithms with even better asymptotic bounds are conceivable. We leave this as an open problem.

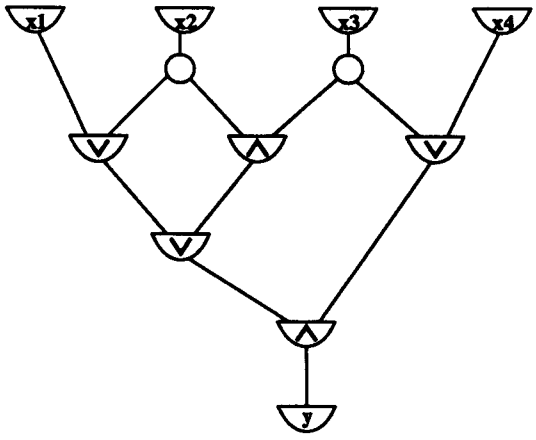

Figure 11: A monotone circuit

4 P-completeness of left-linear semi-unifiability

The existence of a polynomial time sequential algorithm raises the question as to whether there are fast (NC-class) parallel algorithms for left-linear semi-unification. We shall show that, unless NC = P, this is not the case by giving a *log*-space reduction from the circuit value problem for monotone circuits to LLSU, which establishes P-completeness of LLSU under *log*-space reductions.

A *monotone circuit* is a directed acyclic graph $C = (V, E)$ whose nodes, called *gates*, are of five different kinds:

1. *input gates* with no inedge and 1 outedge;
2. *and-gates* with 2 inedges and 1 outedge;
3. *or-gates* with 2 inedges and 1 outedge;
4. *fan-out gates* with 1 inedge and 2 outedges;
5. a single *output gate* with 1 inedge and no outedge.

Furthermore, all gates are reachable from the input gate, and the output gate is reachable from all gates.

Example:

An example of a monotone circuit implementing the Boolean function $y = ((x_1 \vee x_2) \vee (x_2 \wedge x_3)) \wedge (x_3 \vee x_4)$ is shown in Figure 11. The circles represent fan-out gates.

End of example

Every assignment a of truth values to the input gates of C can be extended uniquely to a truth value assignment $\bar{a}$ to all gates of C by defining

1. if n is an input gate, then $\bar{a}(n) = a(n)$;
2. if n is an and-gate with predecessors n', n'', then $\bar{a}(n) = \bar{a}(n') \wedge \bar{a}(n'')$;
3. if n is an or-gate with predecessors n', n'', then $\bar{a}(n) = \bar{a}(n') \vee \bar{a}(n'')$;

4. if n is a fan-out gate with predecessor n' then $\bar{a}(n) = \bar{a}(n')$;

5. if n is the output gate with predecessor n' then $\bar{a}(n) = \bar{a}(n')$.

The *monotone circuit value problem (MCVP)* is the problem of deciding, given monotone circuit C and assignment a to the input gates of C, whether $\bar{a}(n) =$ **true** for the output gate n.

Theorem 6 *LLSU is P-complete under* log*-space reductions.*

Proof:

We give a *log*-space reduction of MCVP to LLSU by adapting a proof of Dwork, Kanellakis, and Mitchell for P-completeness of unification. MCVP is known to be P-complete [Gol77]. Together with the existence of a polynomial time algorithm (see previous section) for LLSU this implies the theorem.

First we describe how we represent a circuit C by a term graph A. Then the assignment a is encoded by adding arrows to A to make it an arrow graph. The thus constructed A will be a reduced arrow graph representation of a left-linear system of inequalities. It is easy to construct the actual system of inequalities instead of its reduced arrow graph representation, but for expositional purposes it is easier to describe the construction of A.

For every kind of gate we describe a term graph "gadget" for that gate. Every one of these gadgets is actually a term graph with a pair of designated nodes for every in- and outedge of the encoded gate. These gadgets are then "glued" together at their input and output node pairs with some arrows to represent C. Additional arrows encode a.

1. An *input gadget* consists of two variable labeled nodes n, n'. It has no input node pair, and its only pair of output nodes is (n, n').

2. An *and-gadget* consists of three nodes n, n', n''. The two pairs $(n, n'), (n', n'')$ are its input node pairs, and (n, n'') is its output node pair.

3. An *or-gadget* consists of two variable labeled nodes n, n'. The two identical pairs (n, n') and (n, n') are its input node pairs, and (n, n') is also its output node pair.

4. A *fan-out gadget* consists of six nodes $n, n_1, n_2, n', n'_1, n'_2$, where n_1, n_2 are the variable labeled children of n and n'_1, n'_2 are the variable labeled children of n' (i.e., n and n' are f-labeled). The input pair is (n, n') and the output pairs are $(n_1, n'_1), (n_2, n'_2)$.

5. The output gate is represented by three nodes n, n', n'', where n', n'' are the variable labeled children of n, and (n, n') is the designated input pair of the gadget. (It has no output pair.)

For a given combinational circuit C we use one of the gadgets above for every gate and connect the input and output node pairs with arrows whenever two gates are connected via an edge. Specifically, if there is an edge from gate g to gate g' in C, the edge corresponds to an output pair in the gadget for g and an input pair in the gadget for g'. Let (n, n') be this output pair in the gadget for g and (m, m') the corresponding input pair in the gadget for g'. We add the arrows $m \to n$ and $n' \to m$. Finally, for every input gate g that is set to **true** by a we add an arrow $n \to n'$ between the output pair (n, n') in the gadget representing g.

If we associate with every arrow in the thus constructed arrow graph A a distinct color then A is an arrow graph representation of a left-linear system of inequalities (that we can construct from A in logarithmic space). It is easy to check that after applying Algorithm L1 to A (ignoring the colors) the resultant arrow graph A' has an arrow path from n to its child n' in the output gadget of A' if and only if the value **true** is assigned to the output gate of C under

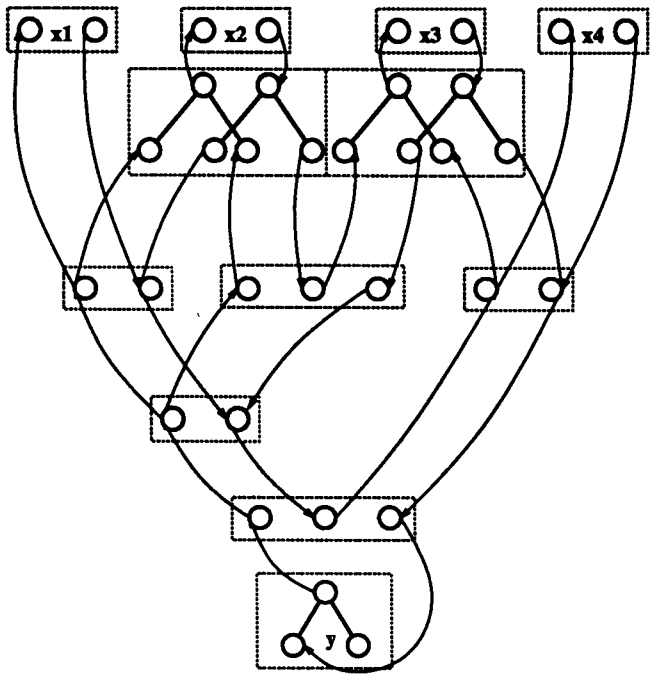

Figure 12: An arrow graph encoding of a monotone circuit

a. Consequently A' fails the acyclicity test if and only if $\bar{a}$ assigns **true** to the output gate in C. Since the construction of A can be implemented in logarithmic space, this proves the theorem.

End of proof

Example:

An arrow graph encoding of the circuit of Figure 11 is shown in Figure 12.

End of example

Let kLLSU be the problem of left-linear semi-unification with exactly k term inequalities. The proof of the previous theorem can be strengthened to yield the following result.

Corollary 2 *2LLSU is P-complete.*

Proof:

Consider the step in the proof of Theorem 6 where a ***distinct*** color is associated with every arrow in the arrow graph A constructed from a monotone circuit C and assignment a. We can make do by using only two colors; in fact, only when connecting the input pairs of an or-gate g we use two distinct colors for the two outarrows from node n in the input pair (n, n') in the or-gadget. For all other arrows we can use the same color. The resulting arrow graph has two colors and represents a left-linear system of 2 inequalities that can be constructed in logarithmic space from the arrow graph.

End of proof

This implies, of course, that kLLSU is P-complete for all $k \geq 2$. We conjecture that 1LLSU is also P-complete.

5 Concluding remarks and open problems

Kfoury, Tiuryn and Urzyczyn showed that left-linear semi-unifiability (LLSU) is decidable. Their proof is essentially an exponential-time decision procedure and thus their result shows that LLSU is in DEXPTIME.

In this paper we have given a tight structural upper and lower bound for LLSU by proving that it is P-complete. We have shown that a dynamic transitive closure based implementation of a generic algorithm yields an $O(n^3)$ time decision procedure for LLSU.

Several issues and open problems remain:

- We conjecture there is an $O(n^2)$ time algorithm based on our generic Algorithm L1.
- We conjecture that 1LLSU (left-linear semi-unifiability with only 1 inequality) is P-complete (we proved completeness for k inequalities for $k > 1$).
- Possible applications of LLSU to proof theory remain to be explored (suggested by Hans Leiß).
- Several generalizations are not directly addressed, but do not pose any major difficulties. By adding a unification-like postprocessing phase to algorithm L1 it is possible to generalize algorithm L1 to arbitrary alphabets (instead of only $\mathcal{A}_2$). A dynamic transitive closure based implementation still takes only $O(n^3)$ time. Furthermore, it is possible to extract a most general semi-unifier from the output of Algorithm L1. In particular, size and algebraic properties of most general semi-unifiers of left-linear systems of inequalities appear to follow immediately from the correctness of L1.

6 Acknowledgements

I wish to thank Hans Leiß for extensive discussions on semi-unification and, in particular, for his probing questions on the correctness of Algorithm L1. I am also thankful to Han La Poutré for showing me how his and Jan van Leeuwen's dynamic transitive closure algorithm can be adapted elegantly to Algorithm L1.

References

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[DR89] J. Doerre and W. Rounds. On subsumption and semiunification in feature algebras. Manuscript, Oct. 1989.

[Gol77] L. Goldschlager. The monotone and planar circuit value problems are log-space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hen88a] F. Henglein. Algebraic properties of semi-unification. Technical Report (SETL Newsletter) 233, New York University, November 1988.

[Hen88b] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming*. ACM, ACM Press, July 1988.

[Hen89] F. Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989.

[Hoo65] P. Hooper. *The Undecidability of the Turing Machine Immortality Problem.* PhD thesis, Harvard University, June 1965. Computation Laboratory Report BL-38.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[KMNS89] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proc. Foundations of Software Technology and Teoretical Computer Science*, Jan. 1989.

[KTU89a] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, June 1989.

[KTU89b] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. Technical Report BUCS-89-010, Boston University, Oct. 1989.

[Lei89a] H. Leiß. Decidability of semi-unification in two variables. Technical Report INF-2-ASE-9-89, Siemens, July 1989.

[Lei89b] H. Leiss. Semi-unification and type inference for polymorphic recursion. Technical Report INF2-ASE-5-89, Siemens, Munich, West Germany, 1989.

[LPvL87] J. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proc. Int'l Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Springer-Verlag, June 1987. Lecture Notes in Computer Science, Vol. 314.

[Pud88] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[Pur87] P. Purdom. Detecting looping simplifications. In *Proc. 2nd Conf. on Rewrite Rule Theory and Applications (RTA)*, pages 54–62. Springer-Verlag, May 1987.

[Yel88] D. Yellin. A dynamic transitive closure algorithm. Technical Report RC 13535, IBM T.J. Watson Research Ctr., June 1988.

Fast left-linear semi-unification

F. Henglein

RUU-CS-89-33
December 1989

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Fast left-linear semi-unification

F. Henglein

Technical Report RUU-CS-89-33
December 1989

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

Fast Left-Linear Semi-Unification

Fritz Henglein
Department of Computer Science
Utrecht University
PO Box 80.089
3508 TB Utrecht
The Netherlands
Internet: henglein@cs.ruu.nl

December 22, 1989

Abstract

Semi-unification is a generalization of both unification and matching with applications in proof theory, term rewriting systems, polymorphic type inference, and natural language processing. It is the problem of solving a set of term inequalities $M_1 \leq N_1, \ldots, M_k \leq N_k$, where $\leq$ is interpreted as the subsumption preordering on (first-order) terms. Whereas the general problem has recently been shown to be undecidable, several special cases are decidable.

Kfoury, Tiuryn, and Urzyczyn proved that *left-linear semi-unification (LLSU)* is decidable by giving an exponential time decision procedure. We improve their result as follows.

1. We present a generic polynomial-time algorithm L1 for LLSU, which shows that LLSU is in P.
2. We show that L1 can be implemented in time $O(n^3)$ by using a fast dynamic transitive closure algorithm.
3. We prove that LLSU is P-complete under *log*-space reductions, thus giving evidence that there are no fast (NC-class) parallel algorithms for LLSU.

As a corollary of the proof of P-completeness we obtain that LLSU with only 2 term inequalities is already P-complete.

We conjecture that L1 can be implemented in time $O(n^2)$, which is the best that is possible for the solution method described by L1. The basic question as to whether another solution method may admit even faster algorithms is open. We conjecture also that LLSU with 1 inequality is also P-complete.

1 Introduction

Semi-unification is a generalization of both unification and matching with applications in proof theory [Pud88], term rewriting systems [Pur87, KMNS89], polymorphic type inference [Hen88b, KTU89a, Lei89b], and natural language processing [DR89]. Because of its fundamental nature it may be expected to find even more applications.

Whereas general semi-unification was long believed to be decidable, Kfoury, Tiuryn and Urzyczyn recently gave an elegant reduction of the *boundedness* problem for deterministic Turing Machines to semi-unification [KTU89b]. By adapting a proof for a similar problem attributed to

Hooper [Hoo65] they showed that boundedness is undecidable, which implies the undecidability of semi-unification.

Several special cases of semi-unification have been shown to be decidable: uniform semi-unification (solving a single term inequality) [Hen88b, Pud88, KMNS89], semi-unification over two variables [Lei89a], and left-linear semi-unification [KTU89a]. Pudlak showed that general semi-unification can be effectively reduced to semi-unification over two inequalities [Pud88]; thus bounding the number of inequalities by a number greater than one does not simplify the problem. In drastic contrast, Kapur, Musser, Narendran and Stillman gave a polynomial time procedure for uniform semi-unifiability (single inequality) [KMNS89].

In this article we investigate the special case of *left-linear semi-unification (LLSU)*. This problem was first addressed by Kfoury, Tiuryn, and Urzyczyn [KTU89a]. They were able to show that left-linear semi-unification is decidable, and a closer look at their decision procedure reveals that left-linear semi-unification is in DEXPTIME. We improve this result by showing that left-linear semi-unification is polynomial time decidable. The present a generic algorithm, Algorithm L1, for LLSU that is implementable in polynomial time. An implementation based on a fast dynamic transitive closure algorithm [LPvL87, Yel88] (with only edge additions) yields an $O(n^3)$ time LLSU procedure where n is the size of the given (left-linear) semi-unification problem. Dynamic transitive closure seems too general and powerful a method for LLSU, and we conjecture that L1 can be improved to run in time $O(n^2)$. This is best possible for any method based on L1 since as many as n^2 edges are added to an initially sparse graph on n nodes in L1. The question as to whether there is a linear-time algorithm for left-linear semi-unification or, in fact, any algorithm asymptotically faster than $O(n^2)$ is left open.

We also show that, even though left-linearity is a very strong condition on input instances, it still is not strong enough to admit fast (NC-class) parallel algorithms unless NC = P. Specifically, we show that LLSU is P-complete under *log*-space reductions by adapting a well-known proof of Dwork, Kanellakis, and Mitchell for showing the P-completeness of unification [DKM84].

The outline of the rest of the paper is as follows. In section 2 we define general and left-linear semi-unification, and we present a general semi-unification algorithm, algorithm A [Hen89]. Observing the behavior of algorithm A on left-linear problem instances yields the critical insight that permits "speeding up" A to run in polynomial time. The result is Algorithm L1. In section 3 we present our polynomial time algorithm L1 for LLSU over the alphabet $\mathcal{A}_2$, which consists of a single binary function symbol, and show that a dynamic transitive closure based implementation has time complexity $O(n^3)$. In section 4 we show that left-linear semi-unification is P-complete. We conclude with some final remarks and open problems in section 5.

2 General semi-unification

In this section we present definition and properties of semi-unification and an (semi-)algorithm, Algorithm A, for solving general semi-unification problem instances. Most of this is a rehash from a previous paper [Hen89], but it is very instructive in giving insight into the correctness of our polynomial time algorithm for LLSU.

2.1 Definition and properties of semi-unification

A *ranked alphabet* $\mathcal{A} = (F, a)$ is a finite set F of *function symbols* together with an *arity* function a that maps every element in F to a natural number (including zero). A function symbol with arity 0 is also called a *constant*. The set of *variables* V is a denumerable infinite set disjoint from F. The *terms* over A and V is the set $T(\mathcal{A}, V)$ consisting of all strings generated by the grammar

$$M ::= x|c|f(\underbrace{M,\ldots,M}_{k\text{times}})$$

where f is a function symbol from $\mathcal{A}$ with arity $k > 0$, c is a constant, and x is any variable from V. Two terms M and N are equal, written as $M = N$, if and only if they are identical as strings.

A *substitution* σ is a mapping from V to $T(\mathcal{A}, V)$ that is the identity on all but a finite subset of V. The set of variables on which σ is *not* the identity is the *domain* of σ. Every substitution $\sigma : V \to T(\mathcal{A}, V)$ can be naturally extended to $\sigma : T(\mathcal{A}, V) \to T(\mathcal{A}, V)$ by defining

$$\sigma(f(M_1,\ldots,M_k)) = f(\sigma(M_1),\ldots,\sigma(M_k)).$$

A term M *subsumes* N (or N *matches* M), written $M \leq N$, if there is a substitution ρ such that $\rho(M) = N$. If N matches M then there is exactly one such ρ as required in the above definition whose domain is contained in the set of variables occurring in M. We call it the *quotient* substitution of M and N and denote it by N/M.

Given a set of pairs of terms $S = \{(M_1, N_1),\ldots,(M_k, N_k)\}$ a substitution σ is a *semi-unifier* of S if $\sigma(M_1) \leq \sigma(N_1),\ldots,\sigma(M_k) \leq \sigma(N_k)$. S is *semi-unifiable* if it has a semi-unifier. Semi-unifiability is reminiscent of both unification (because of σ being applied to both the left- and right-hand components of S) and matching (because the resultant right-hand sides have to match their corresponding right-hand sides), but it is in fact much more general than both, evidenced by the recent undecidability result for semi-unifiability.

In the context of semi-unification we shall call a set of pairs of terms a *system of inequalities* and write it also

$$\begin{array}{rcl} M_1 & \overset{?}{\leq} & N_1 \\ M_2 & \overset{?}{\leq} & N_2 \\ & \cdots & \\ M_k & \overset{?}{\leq} & N_k \end{array}$$

The question mark over the inequality symbol is to indicate that these are not *valid* inequalities, but are supposed to be *solved*; that is, a substitution is to be found that makes them valid.

As shown in [Hen88a], every semi-unifiable system of inequalities has a most general semi-unifier if the notion of generality on substitutions is properly defined.

Example:

Let $\mathcal{A}_2$ be a ranked alphabet with only one function symbol f, which has arity 2; $V = \{x_1, x_2,\ldots,x_i,\ldots\}$.

Consider the semi-unifiable system of inequalities S_0:

$$\begin{array}{rcl} f(f(x_1,x_2),x_3) & \overset{?}{\leq} & x_4 \\ x_4 & \overset{?}{\leq} & f(x_3,f(x_2,x_5)) \end{array}$$

Possible semi-unifiers are $\sigma_0 = \{x_3 \mapsto f(x_6,x_7), x_4 \mapsto f(f(x_8,x_9),f(x_{10},x_{11}))\}$ and $\sigma_1 = \{x_3 \mapsto f(x_2,x_2), x_4 \mapsto f(f(x_2,x_2),f(x_2,x_2)), x_5 \mapsto x_2\}$ (σ_0 is a most general semi-unifier of S_0) since, after substituting σ_0, respectively σ_1, in S_0, we get the valid inequalities

$$\begin{array}{rcl} f(f(x_1,x_2),f(x_6,x_7)) & \leq & f(f(x_8,x_9),f(x_{10},x_{11})) \\ f(f(x_8,x_9),f(x_{10},x_{11})) & \leq & f(f(x_6,x_7),f(x_2,x_5)), \end{array}$$

respectively

$$\begin{aligned} f(f(x_1,x_2),f(x_2,x_2)) &\leq f(f(x_2,x_2),f(x_2,x_2)) \\ f(f(x_2,x_2),f(x_2,x_2)) &\leq f(f(x_2,x_2),f(x_2,x_2)). \end{aligned}$$

The quotient substitutions for these inequalities are $\rho_0^1 = \{x_1 \mapsto x_8, x_2 \mapsto x_9, x_6 \mapsto x_{10}, x_7 \mapsto x_{11}\}$ and $\rho_0^2 = \{x_8 \mapsto x_6, x_9 \mapsto x_7, x_{10} \mapsto x_2, x_{11} \mapsto x_5\}$, respectively $\rho_1^1 = \{x_1 \mapsto x_2\}$ and $\rho_1^2 = \{\}$.

End of example

2.2 Algorithm A

Algorithm A is a general (semi-)algorithm fqr computing the most general semi-unifier of a system of inequalities. Even though it is bound not to terminate for some non-semi-unifiable inputs it catches many non-semi-unifiable systems of inequalities due to an *extended occurs check*, which is a generalization of the conventional occurs check in unification algorithms.

The algorithm operates on a graph-theoretic representation of systems of inequalities, both to achieve practically acceptable performance and to aid in the analysis of some combinatorial properties. Since intermediate steps of the algorithm can introduce *equations* $M \stackrel{?}{=} N$ between terms, not just inequalities, the representation, called an *arrow graph*, in fact encodes systems of equations and inequalities. Because a formal description of arrow graphs is notoriously cumbersome, we give a brief, but hopefully clear, informal definition below.

A *term graph* is an acyclic graph that represents sets of terms over a given alphabet $\mathcal{A} = (F, a)$ and set of variables V. It consists of a set of nodes, N, a subset of which is labeled with function symbols from F, and the rest of which is labeled with variables from V. If f is a function symbol with arity k, $k \geq 0$, every node n labeled with f has exactly k *ordered children*; i.e., there are k directed *term edges* originating in n and labeled with the numbers 1 through k. The variable labeled nodes have no children, and for every variable x there is at most one node labeled with x. The nodes together with the tree edges form a normal directed graph, and if it is acyclic, then we say the term graph is *acyclic*.

Every node in an acyclic term graph can be interpreted as a term; for example, if n is a node labeled with function symbol f, and its children are $n_1, \ldots, n_k$ (in this order) representing terms $M_1, \ldots, M_k$, then n represents the term $f(M_1, \ldots, M_k)$. Note that for every set of terms there is an easily constructed, but generally non-unique term graph such that every term is represented in it.

Example:

The term graph in Figure 1 represents the terms $f(f(x_1,x_2),x_3),x4$ and $f(x_3,f(x_2,x_5))$ occurring in the system of inequalities S_0 (see previous example).

End of example

A term graph can represent all the terms occurring in a system of equations and inequalities. An *arrow graph* is a term graph with two additional *kinds* of edges: *Equivalence edges* encode equations, and *arrows* encode inequalities.

Equivalence edges are represent an equivalence relation on the nodes of the arrow graph. They can be thought of as *undirected edges*, This is the notion we shall adopt, but we shall always assume that for every (undirected) path from node n_1 to node n_2 via equivalence edges (only) there is also an equivalence edge directly between n_1 and n_2. If there is an equivalence edge between n_1 and n_2 we write $n_1 \sim n_2$.

Arrows are directed edges labeled by natural numbers, which indicate to which inequality in a given system of equations and inequalities an arrow corresponds. We call the labels of arrows *colors*, and we write $n_1 \stackrel{i}{\rightarrow} n_2$ if there is an i-colored arrow pointing from n_1 to n_2.

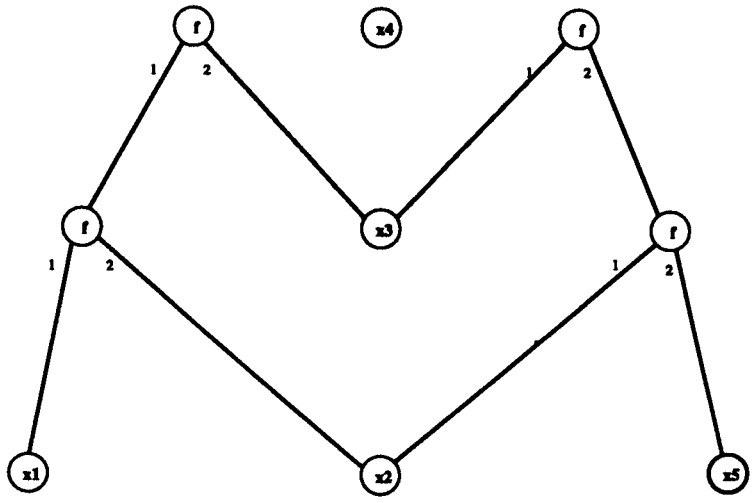

Figure 1: A term graph representation of a set of terms

To summarize, an arrow graph is a term graph with additional edges: undirected equivalence edges and directed arrows. Note that an arrow graph has three different kinds of edges: term edges, equivalence edges, and arrows.

An *arrow graph representation* of a system of inequalities S,

$$\begin{aligned} M_1 &\stackrel{?}{\leq} N_1 \\ M_2 &\stackrel{?}{\leq} N_2 \\ &\cdots \\ M_k &\stackrel{?}{\leq} N_k, \end{aligned}$$

is a term graph G with (not necessarily distinct) nodes $m_1, \ldots, m_k, n_1, \ldots, n_k$ representing the terms in S, and arrows from m_i (representing M_i) to n_i (representing N_i) for $1 \leq i \leq k$ that have pairwise distinct labels. There are no equivalence edges. (In other formulations systems of equations and inequalities are input instances for semi-unification, in which case equivalence edges are used to represent equations in the arrow graph representation.)

Example:

An arrow graph representation of system of inequalities S_0 is in Figure 2. Term edges are shown in straight, bold lines; arrows in curved, thin lines. There are no equivalence edges.

End of example

Algorithm A operates on arrow graphs. It repeatedly rewrites the arrow graph representation of a system of inequalities S by nondeterministically "applying" some closure rules until no rule can be applied any more. At that point it indicates whether S is semi-unifiable and, if so, outputs a most general semi-unifier of S. The algorithm is described in detail in Figure 3 for alphabet $\mathcal{A}_2$[1], and the closure rules are also depicted graphically in Figure 4.

Example:

[1] $\mathcal{A}_2$ is the alphabet consisting of a single binary function symbol f. Semi-unifiability over any alphabet is *log*-space reducible to semi-unifiability over $\mathcal{A}_2$ [Hen89]. This reduction does not, however, preserve left-linearity.

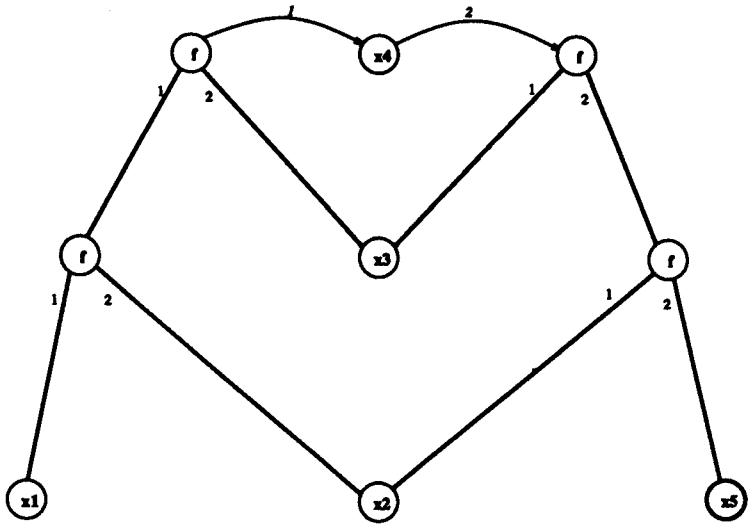

Figure 2: An arrow graph representation of a system of inequalities

The effect of Algorithm A on the arrow representation of the system of inequalities S_0 is shown in Figures 5 and 6. In the initial arrow graph representation (see Figure 2) only rule 4b in Algorithm A (see Figure 3) is applicable. The arrow graph after after its application is shown in Figure 5. Immediately afterwards rule 3a can be applied to add an arrow (colored with 1) pointing to the new f-labeled node, and application of rule 2 adds 1-colored arrows pointing to the new variable-labeled nodes, at which point rule 4b is applicable again. The algorithm finally terminates with the arrow graph shown in Figure 6. Since this final arrow graph is not $\square$ the given system of inequalities is semi-unifiable, and a most general semi-unifier can be read off from the final arrow graph: $\{x_3 \mapsto f(x_9, x_{12}), x_4 \mapsto f(f(x_8, x_{11}), f(x_{10}, x_{13}))\}$, which is equivalent to σ_0 modulo renaming of variables.

End of example

2.3 Left-linear semi-unification

A term is *linear* if every variable has at most one occurrence in it. For example, $f(x_1, x_2)$ is linear (assuming $x_1 \neq x_2$), but $f(x_1, x_1)$ is not. A system of inequalities S is *left-linear* if every left-hand side term in S is linear. ***Left-linear semi-unification*** is the problem of computing most general semi-unifiers of left-linear systems of inequalities, and ***left-linear semi-unifiability (LLSU)*** is the problem of deciding whether a given left-linear system of inequalities is semi-unifiable.

Example:

Recall the system of inequalities S_0,

$$\begin{aligned} f(f(x_1, x_2), x_3) &\stackrel{?}{\leq} x_4 \\ x_4 &\stackrel{?}{\leq} f(x_3, f(x_2, x_5)). \end{aligned}$$

The left-hand sides are the terms $f(f(x_1, x_2), x_3)$ and x_4. Since they are linear, S_0 is an instance of left-linear semi-unification (over alphabet $\mathcal{A}_2$).

Let G be an arrow graph. Apply the following rules (depicted also in Figure 4) until convergence:

1. If there exist nodes m and n labeled with f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \sim n$ then add $m_1 \sim n_1$ and $m_2 \sim n_2$.
2. If there exist nodes m and n labeled with f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \xrightarrow{i} n$ then add arrows $m_1 \xrightarrow{i} n_1$ and $m_2 \xrightarrow{i} n_2$.
3. If there exist nodes m_1, m_2, n_1, and n_2 such that
   (a) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $n_1 \xrightarrow{i} n_2$ then add $m_2 \sim n_2$;
   (b) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $m_2 \sim n_2$ then add an arrow $n_1 \xrightarrow{i} n_2$.
4. (a) (Extended occurs check) If there is a path consisting of arrows of any color (arrow path) from n_1 to n_2 and n_2 is a proper descendant of n_1, then reduce to the improper arrow graph $\square$.[a]
   (b) If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with f and has children m_1, m_2, n is labeled with a variable, and $n \sim n''$ implies that n'' is variable labeled, and there is an arrow $m \xrightarrow{i} n$ then create new nodes n', n'_1, n'_2 and label n' with f, label n'_1 and n'_2 with new variables x' and x'', respectively; make n'_1, n'_2 the children of n'; and add $n \sim n'$.

[a] Node n' is a *descendant* of node n if it there is a path from n to n' consisting of term edges (traversed in forward direction) and equivalence edges (traversed in any direction); n' is a *proper* descendant if there is a path with at least one term edge.

Figure 3: Algorithm A

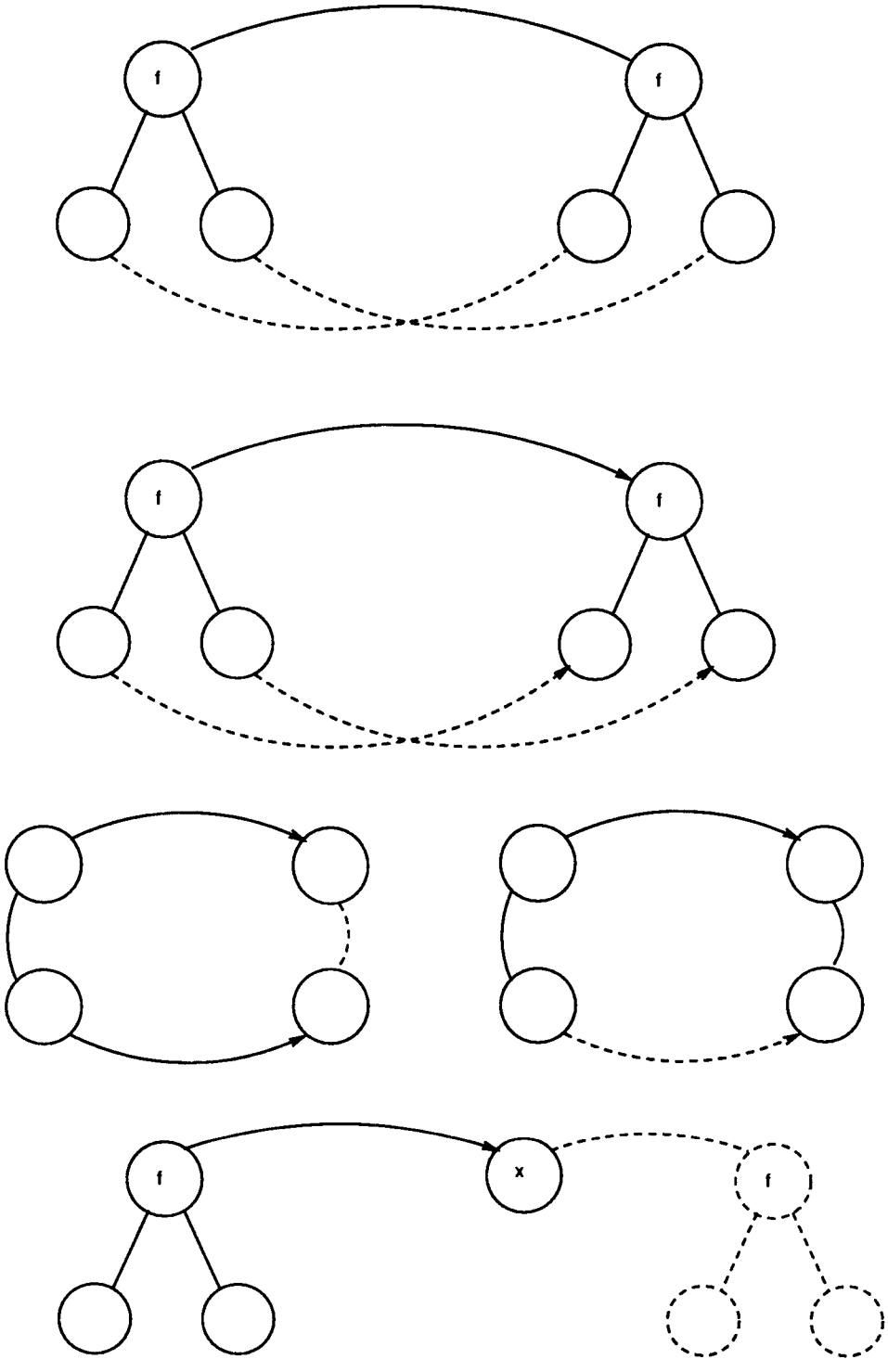

Figure 4: Closure rules

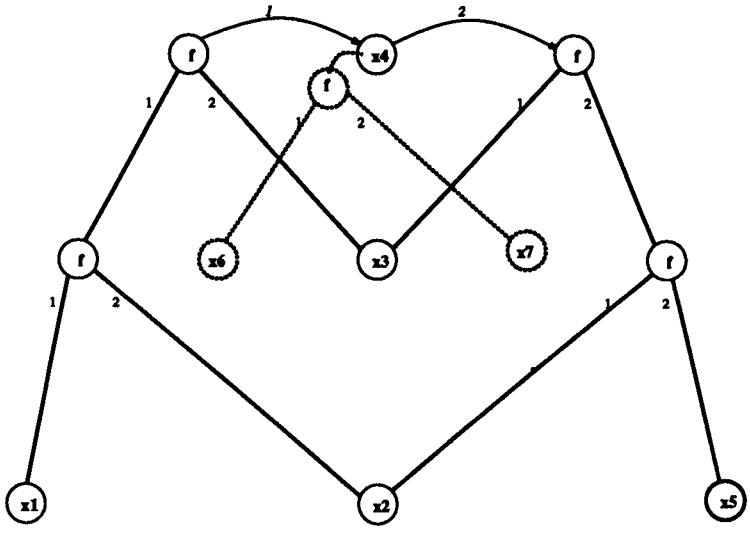

Figure 5: After application of a closure rule

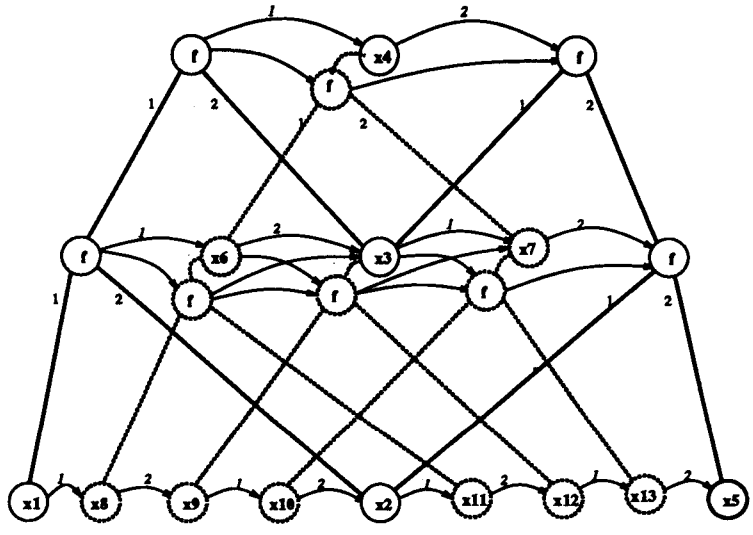

Figure 6: A final arrow graph

(4b)

If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with f and has children m_1, m_2, n is labeled with a variable, and $n \sim n'$ implies that n' is variable labeled, and there is an arrow $m \xrightarrow{i} n$ then create new nodes n'_1, n'_2 and change the label of n to be f, label n'_1 and n'_2 with new variables x' and x'', respectively; and make n'_1, n'_2 the children of n.

Figure 7: Modified rule for expanding arrow graph

End of example

For reasons that will become clear below we can modify rule 4b in an almost trivial fashion. If rule 4b is applicable because there is an arrow from an f-labeled node m to a variable labeled node n, it prescribes that a new f-labeled node n' with new variable labeled children n'_1, n'_2 and an equivalence edge $n \sim n'$ be added to the arrow graph. The *modified* rule 4b is as follows. Instead of adding three new nodes n', n'_1, n'_2 and an equivalence edge $n \sim n'$ we add only two new nodes, n'_1, n'_2 (labeled with new variables), *change* the label of n from a variable to f, and make n'_1, n'_2 the children of n. (The old label of n is remembered in some auxiliary data structure. This information is only needed for reading off the most general semi-unifier from a final arrow graph, but not for the execution of Algorithm A itself.) This modification is given in detail in Figure 7. Henceforth, we shall assume that Algorithm A uses the modified rule 4b.

The correctness of Algorithm A is proved in great detail in [Hen89], and it is a minor issue to verify that the modified version of rule 4b preserves correctness.

Example:

The final arrow graph after executing modified Algorithm A on the arrow graph representation of system S_0 of Figure 2 is shown in Figure 8. Note that the arrow graph contains no equivalence edges, and only rules 2 and 4b of Algorithm A were applied.

End of example

Let us consider an arrow graph representation of a left-linear system of inequalities S and one of its nodes m that represents a left-hand side. Since S is left-linear the subgraph rooted at m is a tree; that is, there is one and only one path consisting of term edges from m to any of the nodes reachable from m via term edges. Executing Algorithm A on the arrow graph of Figure 2 we see that at no point rule 3a, the only rule that could possibly introduce the first equivalence edge, is applicable. This is not just a peculiar property of the specific example, but holds for every (arrow representation of a) left-linear system of inequalities.

An *execution (of Algorithm A)* is a sequence of arrow graphs $(G_0, \ldots, G_i, \ldots)$ in which every component is derived from its predecessor by application of one of the closure rules of Algorithm A (see Figure 3 or Figure 4).

Theorem 1 *Let $(G_0, \ldots, G_i, \ldots)$ be an execution of Algorithm A. If G_0 is an arrow graph representation of a left-linear system of inequalities, then G_{i+1} is derived from G_i by rule 2 or rule 4.*

Proof:

For $(G_0, \ldots, G_i, \ldots)$ we can prove by induction on i that the following properties hold for every $G_i \neq \Box$:

1. Every node has at most one outarrow of any given color.

2. The subgraph rooted at any node with an outarrow (of any color) is a tree

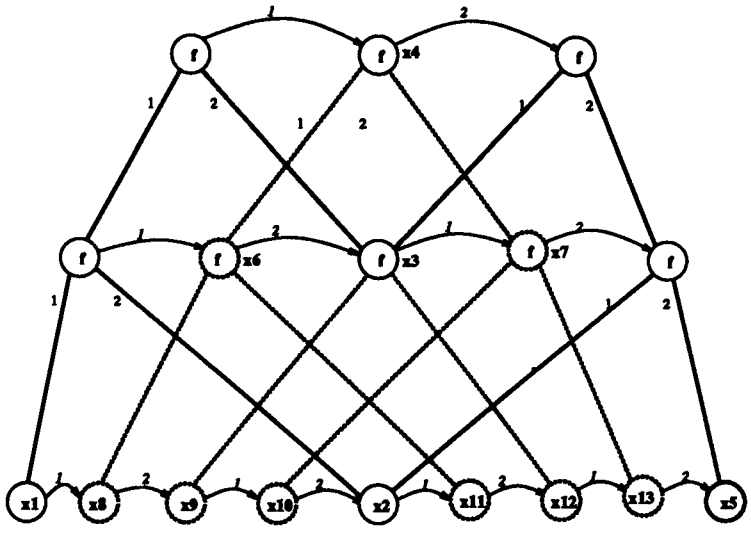

Figure 8: A final arrow graph produced by modified Algorithm A

This implies the theorem since it guarantees that neither rule 3 nor rule 1 is applicable to any of the G_i's.

End of proof

This theorem yields the critical insight for speeding up Algorithm A for the special case of solving left-linear semi-unification problems. We will give very informal considerations below that will lead us, directly from the observation of theorem 1, to the polynomial time LLSU-algorithm L1 below. L1 is presented in and its correctness is proved in the following section.

A quite immediate simplification of Algorithm A is that, for left-linear semi-unification, rules 3 and 1 are not needed. But further simplifications are possible. Note that the color maintained with every arrow is only needed as a criterion for applying rule 3. For example, if a node n has two outarrows with *equal* color to distinct nodes n' and n'' an equivalence edge $n' \sim n''$ has to be added, but if the two arrows have *different* colors no such equivalence edge has to be added. Because, by theorem 1, the first case can never happen for left-linear systems of inequalities, we can dispense with the coloring information on arrows altogether.

But without color information we may also maintain the arrows transitively closed; that is, we can maintain an arrow from n_1 to n_2 (all arrows are uncolored now) with every arrow *path* from n_1 to n_2. This is not an advantage by itself, but it does away with the need to apply rule 4b at all, if we adopt a modified extended occurs check rule (rule 4a). Basically, the only relevant effect of rule 4b on left-linear arrow graph representations is to create establish arrow paths between the children of f-labeled nodes that are in turn connected with an arrow path. This can be achieved directly, without adding new nodes, by maintaining the transitive closure of the arrows as indicated above or by generalizing rule 2 to apply arrow paths instead of only individual arrows. If there is an arrow path from a function symbol labeled node n_1 to another function symbol labeled node n_2 in the original algorithm, our strategy of maintaining an arrow

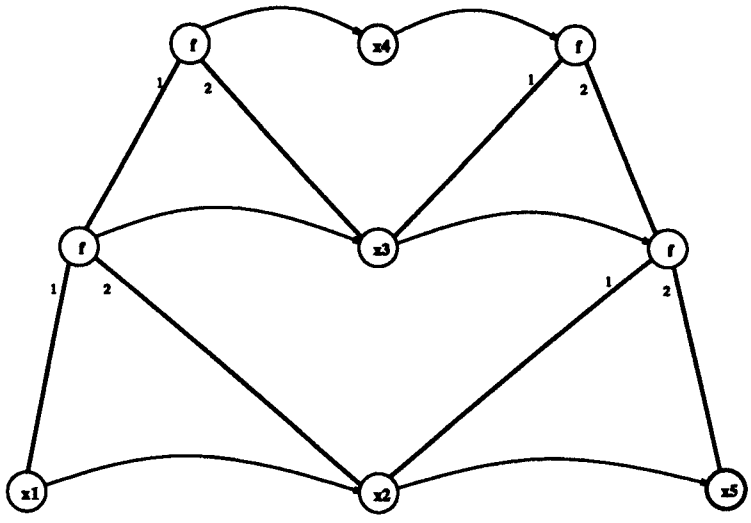

Figure 9: A final arrow with "shortcutting"

for every arrow path guarantees that there is also an arrow from n_1 to n_2, and applying rule 2, which propagates arrows downwards to the children of n_1 and n_2, will guarantee that there is an arrow from any child of n_1 to the corresponding child of n_2. We call the "contraction" of arrow paths to single arrows *shortcutting* since repeated copying with rule 4b is not necessary any more — it is "shortcut".

Example:

Consider again the system of inequalities S_0. The algorithm outlined informally above will simply propagate an arrow *path* between two f-labeled nodes to *single* arrows between their corresponding children with rule 2. Instead of the arrow graph of Figure 8 the we obtain the final arrow graph in Figure 9.

End of example

This leads to a polynomial time algorithm as can be easily seen from the fact that it is not necessary any more to add new nodes and all computational steps — maintaining the transitive closure and applying rule 2 — can be performed in polynomial time.

3 A polynomial time LLSU algorithm

In this section we present a polynomial time algorithm, Algorithm L1, for solving left-linear systems of inequalities over alphabet $\mathcal{A}_2$. In fact, the algorithm can be used to compute a most general semi-unifier, but we shall leave this aspect unexplored at this point. We show that L1 can be made to run in time $O(n^3)$ by adapting the fast dynamic transitive closure algorithm of La Poutré and van Leeuwen [LPvL87].

A *reduced arrow graph* is an arrow graph without equivalence edges and with only one color on arrows, which is consequently completely redundant and may be left off. We write $n \rightarrow n$ for an uncolored arrow from n to n'. A reduced arrow graph representation of a system of inequalities is simply an arrow graph representation with the colors on arrows left off.

Our first algorithm, Algorithm L1, is described in Figure 10. Roughly, it starts with a reduced

algorithm $L1(A$: **reduced arrow graph)**
while there exist nodes m_1, n_1 and m_2, n_2 in A and number i such that
n_1 is the i-th child of m_1 and n_2 is the i-th child of m_2,
m_1, m_2 are f-labeled (have the same function symbol label),
there is an arrow path from m_1 to m_2,
and there is no arrow from n_1 to n_2
do
add an arrow (arrow path) from n_1 to n_2 to A
end while;
if there is a cycle in A consisting arrows traversed in *forward* direction
and term edges traversed in *backward* direction (i.e., from child to parent)
then signal non-semi-unifiability
else signal semi-unifiability
end if;

Figure 10: Algorithm L1

arrow graph representation of a given system of inequalities, in which the colors of arrows are ignored. It then looks for arrow paths between f-labeled nodes and adds (uncolored) arrows between their corresponding children until this entails no more changes to the reduced arrow graph. In the final stage, the resultant reduced arrow graph is checked whether it contains a cycle that is made up of arrows — traversed in forward direction — and term edges — traversed from child to parent — and contains at least one term edge. If there is such a cycle we say the final arrow graph *fails* the *acyclicity test* and the algorithm signals non-semi-unifiability; otherwise, the final arrow graph *passes* the acyclicity test, and the algorithm signals semi-unifiability.

We first establish the correctness of L1 and then show that it can be implemented in polynomial time. To address correctness we need to recall some results on the structure of terms with respect to subsumption [Hue80].

Term subsumption is a preordering. The equivalence relation canonically induced by $\leq$ is defined by $M \cong N \Leftrightarrow M \leq N \wedge N \leq M$. We write $[M]$ for the $\cong$-equivalence class of term M; $T(\mathcal{A}, V)/_{\cong}$ for the set of equivalence classes of all terms over A and V; and $\leq$ for the partial order on $T(\mathcal{A}, V)/_{\cong}$ induced by the preorder $\leq$.

Theorem 2 $(T^{\Omega}/_{\cong}, \leq)$ *is a complete lattice.*

Proof: See [Hue80]. **End of proof**

In particular, every set of terms M has a least upper bound $\bigvee \mathcal{M}$ w.r.t. the subsumption preordering $\leq$ that is unique modulo $\cong$. The equivalence relation $\cong$ is also sometimes informally referred to as "renaming of variables". Note that every finite set of terms has a least upper bound whose variables are disjoint from the variables occurring in *any* fixed finite set.

Theorem 3 *Let A be a reduced arrow graph representing a left-linear system of inequalities S. If A signals non-semi-unifiability then S is not semi-unifiable. If A signals semi-unifiability then S is is semi-unifiable.*

Proof:

We shall show that if L1 fails the acyclicity test then S has no semi-unifier, and if L1 passes the acyclicity test then it has a semi-unifier. In fact the most-general semi-unifier can be extracted from the final, acyclic reduced arrow graph.

1. (L1 fails the acyclicity test)

   Assume S has a semi-unifier σ. Then every node n in the final arrow graph represents a unique term $T(n)$ with respect to σ according to the following rule.

$$T_\sigma(n) = \begin{cases} f(T_\sigma(n_1), T_\sigma(n_2)) & \text{if } n \text{ is } f\text{-labeled with children } n_1, n_2 \\ \sigma(x) & \text{if } n \text{ is labeled with variable } x \end{cases}$$

   We shall write $|M|$ for the size of M, measured in terms of the number of function symbol occurrences and variable occurrences in M. If σ is a semi-unifier of S, then it is easy to see that the final reduced arrow graph must satisfy the following relations. If $n \rightarrow n'$ then $|T_\sigma(n)| \leq |T_\sigma(n')|$, and if n is a child of n' then $|T_\sigma(n)| < |T_\sigma(n')|$. But if the final reduced arrow graph contains a cycle with at least one term edge then there is a node n with $|T_\sigma(n)| < |T_\sigma(n)|$, which is impossible. Consequently, S does not have a semi-unifier.

2. (L1 passes the acyclicity test)

   If L1 passes the acyclicity test, the final reduced arrow graph A' is acyclic in the sense that all cycles consist of arrows only. We define a term interpretation T for every node in A' as follows. Consider the strong components $C_1, \ldots, C_k$ of A' in topological order. For every node n in a strong component C_i we associate a term $T(n)$.

   Let $T(n)$ be defined for all nodes in components $C_1, \ldots, C_{i-1}$. Consider the predecessors $n_1, \ldots, n_k$ of node n in component C_i where, for all $j \in \{1, \ldots, k\}$, $n_j \in C_{j'}$ for some $j' < i$, and n is labeled with variable x. If $k = 0$ then $T(n) = x$. Otherwise, define the *preliminary* term interpretation $T_{prelim}(n) = \bigvee\{T(n_1), \ldots, T(n_k)\}$. If n is f-labeled with children n_1, n_2 we define $T_{prelim}(n) = f(T(n_1), T(n_2))$.

   Now, if C_i consists of the nodes $n, n', \ldots, n^{(l)}$, let

   $T = \bigvee\{T_{prelim}(n), T_{prelim}(n'), \ldots, T_{prelim}(n^{(l)})\}$,

   and let $T', \ldots, T^{(l)}$ be $\cong$-equivalent to T with pairwise disjoint variables. Finally, define $T(n) = T, T(n') = T', \ldots, T(n^{(l)}) = T^{(l)}$.

   Note that T is not uniquely defined, but, most importantly, it has the property that for every pair of variable labeled nodes n, n' the sets of variables in $T(n), T(n')$ are disjoint.[2]

   Define substitution $\sigma = \bigcup\{x \mapsto T(n) : n \text{ is labeled with variable } x\}$. By construction of T we have that for every arrow $n \rightarrow n'$ in A', where n' is variable labeled, there is a substitution ρ such that $\rho(T(n)) = T(n')$; in particular, we can take $\rho = T(n')/T(n)$. By induction on the term structure in A' it follows that this holds also for an f-labeled node n'. Since it holds, in particular, for all arrows in the original arrow graph A this shows that σ is a semi-unifier. Some more analysis shows that it is a most general semi-unifier.

End of proof

Theorem 4 *Algorithm L1 is implementable in polynomial time.*

Proof:

Algorithm L1 can add at most n^2 new arrows to an arrow graph of size n, and the applicability of a single arrow addition step takes time at most $O(n^3)$ (with a naive transitive closure algorithm). Finally, the acyclicity testing step can be performed in time $O(n^2)$.

End of proof

[2] We assume that every least upper bound of a finite set of terms defined above has variables disjoint from all variables occurring in all previously defined term interpretations.

Corollary 1 *Left-linear semi-unifiability is in P.*

In the following theorem we show that adapting a fast dynamic transitive closure algorithm can be used to implement Algorithm L1 in time $O(n^3)$.

Theorem 5 *There is a dynamic transitive closure based implementation of Algorithm L1 that runs in $O(n^3)$ time for a left-linear system of inequalities of size n.*

Proof: (Sketch)

Since there can be as many as $O(n^2)$ additions of arrows in Algorithm A, a naive implementation that maintains the transitive closure of all arrows in $O(n^2)$ with respect to a single edge addition takes a total of $O(n^4)$ time. The algorithms of La Poutré/Van Leeuwen [LPvL87] and Yellin [Yel88], however, have accumulative cost of $O(n^3)$ and can be modified to permit finding a pair of f-labeled nodes n, n' whose children are not yet connected via arrows as a by-product of maintaining the transitive closure of the arrow graph. Consequently the total cost of the **while**-loop is $O(n^3)$. The final acyclicity test can be implemented in time $O(n^2)$ with a fast maximal strong components algorithm.

We give a sketch, due to Han La Poutré, of a modified dynamic transitive closure implementation that permits fast execution of the Boolean test in the while-loop of Algorithm L1 (see Figure 10) as part of updates after edge additions. Basically, we work with 5 copies, $V, V_{lc}, V_{rc}, V_{lp}, V_{rp}$, of the original nodes in the arrow graph A. Initially, we add edges as follows.

- If n_l is a left child of n in A then we add an edge from the copy of n_l in V_{lc} to the copy of n in V and an edge from the copy of n in V to the copy of n_l in V_{lp}.
- If n_r is a right child of n in A then we add an edge from the copy of n_r in V_{rc} to the copy of n in V and an edge from the copy of n in V to the copy of n_r in V_{rp}.
- If there is an arrow $n \rightarrow n'$ then we add an edge from the copy of n in V to the copy of n' in V.

Let us call the resulting directed graph G. We maintain the transitive closure of G in a bit matrix (of size $O(n^2)$). The while-loop in Algorithm L1 is executed as follows. Let S be the workset of pairs of nodes n, n' where n is a node in V_{lp} and n' in V_{lc} (or n in V_{rp} and n' in V_{rc}) and n reaches n' in G. These pairs represent the candidates for which an arrow has to be added. While there is a pair (n, n') in S, we delete it from S and, if there is no edge between the copies of n and n' in V (!) we add it, calculate the transitive closure of G, and update S accordingly; i.e., if a new node n' in V_{lc} (V_{rc}) becomes reachable from a node n in V_{lp} (V_{rp}), we add the pair (n, n') to S.

This sketch shows that finding update candidates can be performed as part of a dynamic transitive closure algorithm without additional asymptotic cost. Consequently, using the fast dynamic transitive closure algorithm for edge additions of La Poutré and van Leeuwen [LPvL87] yields an $O(n^3)$ implementation of Algorithm L1.

End of proof

Since the arrow additions are predetermined by Algorithm L1 itself, we believe that there is an $O(n^2)$ implementation of L1 based on a dynamic depth-first search algorithm. Since there can be as many as $O(n^2)$ arrows added by L1, this would be the best that is possible for L1. Of course, other algorithms with even better asymptotic bounds are conceivable. We leave this as an open problem.

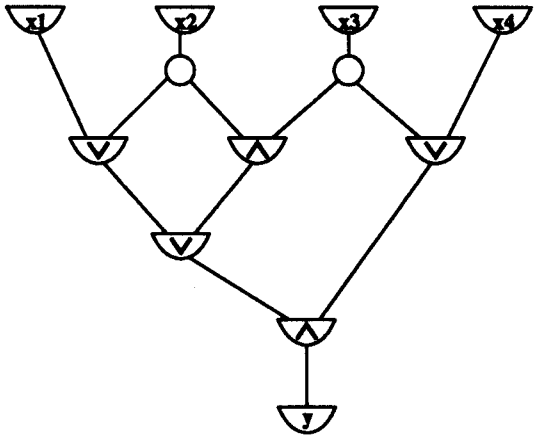

Figure 11: A monotone circuit

4 P-completeness of left-linear semi-unifiability

The existence of a polynomial time sequential algorithm raises the question as to whether there are fast (NC-class) parallel algorithms for left-linear semi-unification. We shall show that, unless NC = P, this is not the case by giving a *log*-space reduction from the circuit value problem for monotone circuits to LLSU, which establishes P-completeness of LLSU under *log*-space reductions.

A *monotone circuit* is a directed acyclic graph $C = (V, E)$ whose nodes, called *gates*, are of five different kinds:

1. *input gates* with no inedge and 1 outedge;
2. *and-gates* with 2 inedges and 1 outedge;
3. *or-gates* with 2 inedges and 1 outedge;
4. *fan-out gates* with 1 inedge and 2 outedges;
5. a single *output gate* with 1 inedge and no outedge.

Furthermore, all gates are reachable from the input gate, and the output gate is reachable from all gates.

Example:

An example of a monotone circuit implementing the Boolean function $y = ((x_1 \vee x_2) \vee (x_2 \wedge x_3)) \wedge (x_3 \vee x_4)$ is shown in Figure 11. The circles represent fan-out gates.

End of example

Every assignment a of truth values to the input gates of C can be extended uniquely to a truth value assignment $\bar{a}$ to all gates of C by defining

1. if n is an input gate, then $\bar{a}(n) = a(n)$;
2. if n is an and-gate with predecessors n', n'', then $\bar{a}(n) = \bar{a}(n') \wedge \bar{a}(n'')$;
3. if n is an or-gate with predecessors n', n'', then $\bar{a}(n) = \bar{a}(n') \vee \bar{a}(n'')$;

4. if n is a fan-out gate with predecessor n' then $\bar{a}(n) = \bar{a}(n')$;

5. if n is the output gate with predecessor n' then $\bar{a}(n) = \bar{a}(n')$.

The *monotone circuit value problem (MCVP)* is the problem of deciding, given monotone circuit C and assignment a to the input gates of C, whether $\bar{a}(n) =$ **true** for the output gate n.

Theorem 6 *LLSU is P-complete under* log*-space reductions.*

Proof:
We give a *log*-space reduction of MCVP to LLSU by adapting a proof of Dwork, Kanellakis, and Mitchell for P-completeness of unification. MCVP is known to be P-complete [Gol77]. Together with the existence of a polynomial time algorithm (see previous section) for LLSU this implies the theorem.

First we describe how we represent a circuit C by a term graph A. Then the assignment a is encoded by adding arrows to A to make it an arrow graph. The thus constructed A will be a reduced arrow graph representation of a left-linear system of inequalities. It is easy to construct the actual system of inequalities instead of its reduced arrow graph representation, but for expositional purposes it is easier to describe the construction of A.

For every kind of gate we describe a term graph "gadget" for that gate. Every one of these gadgets is actually a term graph with a pair of designated nodes for every in- and outedge of the encoded gate. These gadgets are then "glued" together at their input and output node pairs with some arrows to represent C. Additional arrows encode a.

1. An *input gadget* consists of two variable labeled nodes n, n'. It has no input node pair, and its only pair of output nodes is (n, n').

2. An *and-gadget* consists of three nodes n, n', n''. The two pairs $(n, n'), (n', n'')$ are its input node pairs, and (n, n'') is its output node pair.

3. An *or-gadget* consists of two variable labeled nodes n, n'. The two identical pairs (n, n') and (n, n') are its input node pairs, and (n, n') is also its output node pair.

4. A *fan-out gadget* consists of six nodes $n, n_1, n_2, n', n'_1, n'_2$, where n_1, n_2 are the variable labeled children of n and n'_1, n'_2 are the variable labeled children of n' (i.e., n and n' are f-labeled). The input pair is (n, n') and the output pairs are $(n_1, n'_1), (n_2, n'_2)$.

5. The output gate is represented by three nodes n, n', n'', where n', n'' are the variable labeled children of n, and (n, n') is the designated input pair of the gadget. (It has no output pair.)

For a given combinational circuit C we use one of the gadgets above for every gate and connect the input and output node pairs with arrows whenever two gates are connected via an edge. Specifically, if there is an edge from gate g to gate g' in C, the edge corresponds to an output pair in the gadget for g and an input pair in the gadget for g'. Let (n, n') be this output pair in the gadget for g and (m, m') the corresponding input pair in the gadget for g'. We add the arrows $m \to n$ and $n' \to m$. Finally, for every input gate g that is set to **true** by a we add an arrow $n \to n'$ between the output pair (n, n') in the gadget representing g.

If we associate with every arrow in the thus constructed arrow graph A a distinct color then A is an arrow graph representation of a left-linear system of inequalities (that we can construct from A in logarithmic space). It is easy to check that after applying Algorithm L1 to A (ignoring the colors) the resultant arrow graph A' has an arrow path from n to its child n' in the output gadget of A' if and only if the value **true** is assigned to the output gate of C under

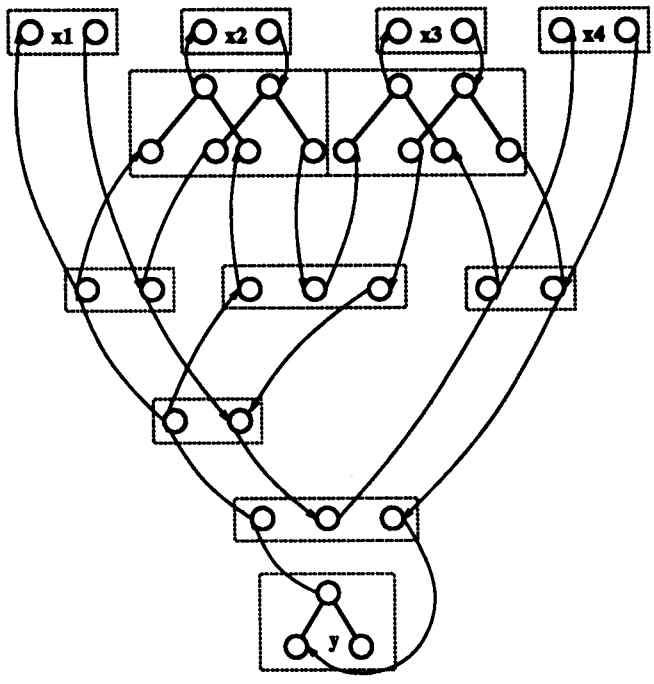

Figure 12: An arrow graph encoding of a monotone circuit

a. Consequently A' fails the acyclicity test if and only if $\bar{a}$ assigns **true** to the output gate in C. Since the construction of A can be implemented in logarithmic space, this proves the theorem.

End of proof

Example:

An arrow graph encoding of the circuit of Figure 11 is shown in Figure 12.

End of example

Let kLLSU be the problem of left-linear semi-unification with exactly k term inequalities. The proof of the previous theorem can be strengthened to yield the following result.

Corollary 2 *2LLSU is P-complete.*

Proof:

Consider the step in the proof of Theorem 6 where a ***distinct*** color is associated with every arrow in the arrow graph A constructed from a monotone circuit C and assignment a. We can make do by using only two colors; in fact, only when connecting the input pairs of an or-gate g we use two distinct colors for the two outarrows from node n in the input pair (n, n') in the or-gadget. For all other arrows we can use the same color. The resulting arrow graph has two colors and represents a left-linear system of 2 inequalities that can be constructed in logarithmic space from the arrow graph.

End of proof

This implies, of course, that kLLSU is P-complete for all $k \geq 2$. We conjecture that 1LLSU is also P-complete.

5 Concluding remarks and open problems

Kfoury, Tiuryn and Urzyczyn showed that left-linear semi-unifiability (LLSU) is decidable. Their proof is essentially an exponential-time decision procedure and thus their result shows that LLSU is in DEXPTIME.

In this paper we have given a tight structural upper and lower bound for LLSU by proving that it is P-complete. We have shown that a dynamic transitive closure based implementation of a generic algorithm yields an $O(n^3)$ time decision procedure for LLSU.

Several issues and open problems remain:

- We conjecture there is an $O(n^2)$ time algorithm based on our generic Algorithm L1.
- We conjecture that 1LLSU (left-linear semi-unifiability with only 1 inequality) is P-complete (we proved completeness for k inequalities for $k > 1$).
- Possible applications of LLSU to proof theory remain to be explored (suggested by Hans Leiß).
- Several generalizations are not directly addressed, but do not pose any major difficulties. By adding a unification-like postprocessing phase to algorithm L1 it is possible to generalize algorithm L1 to arbitrary alphabets (instead of only $\mathcal{A}_2$). A dynamic transitive closure based implementation still takes only $O(n^3)$ time. Furthermore, it is possible to extract a most general semi-unifier from the output of Algorithm L1. In particular, size and algebraic properties of most general semi-unifiers of left-linear systems of inequalities appear to follow immediately from the correctness of L1.

6 Acknowledgements

I wish to thank Hans Leiß for extensive discussions on semi-unification and, in particular, for his probing questions on the correctness of Algorithm L1. I am also thankful to Han La Poutré for showing me how his and Jan van Leeuwen's dynamic transitive closure algorithm can be adapted elegantly to Algorithm L1.

References

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[DR89] J. Doerre and W. Rounds. On subsumption and semiunification in feature algebras. Manuscript, Oct. 1989.

[Gol77] L. Goldschlager. The monotone and planar circuit value problems are log-space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hen88a] F. Henglein. Algebraic properties of semi-unification. Technical Report (SETL Newsletter) 233, New York University, November 1988.

[Hen88b] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming*. ACM, ACM Press, July 1988.

[Hen89] F. Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989.

[Hoo65] P. Hooper. *The Undecidability of the Turing Machine Immortality Problem.* PhD thesis, Harvard University, June 1965. Computation Laboratory Report BL-38.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[KMNS89] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proc. Foundations of Software Technology and Teoretical Computer Science*, Jan. 1989.

[KTU89a] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, June 1989.

[KTU89b] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. Technical Report BUCS-89-010, Boston University, Oct. 1989.

[Lei89a] H. Leiß. Decidability of semi-unification in two variables. Technical Report INF-2-ASE-9-89, Siemens, July 1989.

[Lei89b] H. Leiss. Semi-unification and type inference for polymorphic recursion. Technical Report INF2-ASE-5-89, Siemens, Munich, West Germany, 1989.

[LPvL87] J. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proc. Int'l Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Springer-Verlag, June 1987. Lecture Notes in Computer Science, Vol. 314.

[Pud88] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[Pur87] P. Purdom. Detecting looping simplifications. In *Proc. 2nd Conf. on Rewrite Rule Theory and Applications (RTA)*, pages 54–62. Springer-Verlag, May 1987.

[Yel88] D. Yellin. A dynamic transitive closure algorithm. Technical Report RC 13535, IBM T.J. Watson Research Ctr., June 1988.