RAIRO. INFORMATIQUE THÉORIQUE

HASSAN AÏT-KACI

An algorithm for finding a minimal recursive path ordering

RAIRO. Informatique théorique, tome 19, n° 4 (1985), p. 359-382

<http://www.numdam.org/item?id=ITA_1985__19_4_359_0>

R.A.I.R.O.Informatique théorique/Theoretical Informatics
(vol. 19, n° 4, 1985, p. 359 à 382)

AN ALGORITHM FOR FINDING A MINIMAL RECURSIVE PATH ORDERING (*)

by Hassan Aït-Kaci (1)

Communicated by J. Gallier

Abstract. — *This paper proposes an automatic inference method to compute a minimal partial ordering on a set of function symbols, given a set of term-rewriting rules, which induces a recursive path ordering such that each rule is a strict reduction. A Prolog program is described that implements the method. A direct corollary is the complete automation of the Knuth-Bendix method and a termination-proving program.*

Résumé. — *Ce papier propose une méthode automatique de calcul d'une relation d'ordre minimale sur un ensemble de symbols de fonctions, dans le contexte d'un ensemble de règles de réécriture. Cette relation d'ordre induit un « ordre de chemin recursif » sur les termes tel que toute règle soit une réduction stricte. Un programme en Prolog implantant cette méthode y est décrit. L'automatisation de la méthode de Knuth-Bendix ainsi qu'un programme de preuve de terminaison de la réécriture peuvent ainsi être obtenus en corollaire.*

1.0 INTRODUCTION

A great number of formal systems have made implicit or explicit use of the notion of transformation rules. Computer science has thus systematically given the concept a crucial place in its arsenal of analytic tools. Lately, a soaring and exciting area in the theory of computation has emerged which focuses on the study of a special type of transformational systems; namely, term-rewriting systems. Abstract data type specification [ADJ78], [GOT79], [MUS79], programming language and equational computation theory [ODN78], [HOD82], [HOP80], [PTS81], [ROS73], to name a few, have centered on the concept of rewriting terms.

(*) Received and accepted August 1984.
(1) Department of Computer and Information Science, The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA 19104.

R.A.I.R.O. Informatique théorique/Theoretical Informatics
0399-0540/85/04 359 24/$ 4.40/

A particular method proposed by D. Knuth and P. Bendix in 1970 [KNB70], [HUE81], has been widely used by a majority of researchers in the field. It consists of a "completion algorithm" designed to complete a set of term-rewriting rules to have the most desired property of being "confluent". Roughly, confluence amounts to the possibility of applying any applicable rules in any order to a given term and, if this process terminates, obtaining a unique irreducible term regardless of how and what rules are used. The Knuth-Bendix completion is invaluable for many purposes such as equational theorem-proving and program-synthesis. Unfortunately, the method may never terminate if it is given a non-terminating set of rules. Even worse is its requirement that a term ordering be known *a priori* to orient equations into uni-directional rules.

The first ailment – non-termination – may be averted if a given set of rules can be proven to always terminate. N. Dershowitz [DER82.*a*] defines a particular term ordering which can be used for proving termination of a large class of rewriting systems. He called it *Recursive Path Ordering*, after D. Plaisted's [PLT78] original *Path of Subterms Ordering* on which he based his definition. However, one must already know a well-founded ordering on the set of function symbols in order to find a sufficient condition of termination.

The second inconvenience that a term ordering be provided ahead of time to make use of the Knuth-Bendix algorithm stems from the fact that equations are used implicitly as simplification rules. However, the "simpler" side of an equation must be explicitly specified as right-hand side, rather than being determined so from its structure.

This paper proposes a practical automatic inference method to compute a minimal partial ordering on a set of function symbols, given a set of term-rewriting rules, which induces a recursive path ordering such that each rule be a strict reduction. A Prolog program is described that implements the method. The direct corollary is the complete automation of the Knuth-Bendix method and a termination-proving program.

The following sections 2 and 3 introduce the necessary background on multiset ordering and term-rewriting systems. Next, section 4 presents the algorithm for finding a minimal recursive path ordering. Finally, section 5 describe the Prolog implementation. An appendix shows sample runs of the program.

The author wishes to thank Jean Gallier, Jean-Pierre Jouannaud and the referee for helpful comments.

2.0 MULTISETS AND MULTISET ORDERING

This section introduces the necessary background on multisets and multiset ordering. The latter notion proves itself invaluable as a tool for showing termination of term-rewriting systems [DRM79], as will be seen in the section following this one. A study of the multiset ordering can be found in [JOL82].

DEFINITION 2.1: Let **U** be a non-empty universe of objects. A *multiset* M of objects of **U** is a function from **U** to **N**, the set of non-negative integers.

If M is a multiset on **U**, and $x \in \mathbf{U}$, we write $x \in M$ if $M(x)$ is positive, and the object x is said to be an *element* of M. For any object x in **U**, $M(x)$ is called the *number of occurrences* or *multiplicity* of x in M.

We shall identify a multiset M with the subset of its graph containing all and only those pairs $(x, M(x))$ such that $x \in M$. To make notation as light as possible, we will conventionally omit the multiplicity of an element when it is equal to 1. For example, the multiset loosely written as $\{(a, 2), b, (c, 3), d\}$, is in fact the multiset $\{(a, 2), (b, 1), (c, 3), (d, 1)\}$.

Operations and relations on multisets can be defined and characterized in terms of multiplicity of elements as follows:

Union: $(M_1 \cup M_2)(x) = \max(M_1(x), M_2(x))$;

Intersection: $(M_1 \cap M_2)(x) = \min(M_1(x), M_2(x))$;

Sum: $(M_1 + M_2)(x) = M_1(x) + M_2(x)$;

Difference: $(M_1 - M_2)(x) = \max(0, M_1(x) - M_2(x))$;

Inclusion: $M_1 \subseteqq M_2$ iff $\forall x \in \mathbf{U}$, $M_1(x) \leqq M_2(x)$.

Multisets are also often referred to as "bags" in the literature.

A *strict partial ordering* on a set **S** is a transitive and irreflexive relation on **S**. The following definition extends such a relation on objects to one on multisets of objects.

DEFINITION 2.2: Let **U** be a non-empty universe. Any strict partial ordering $>$ on the elements of **U** can be extended to an ordering $\gg$ on the set of multisets of **U** as follows: $M_1 \gg M_2$ iff (1) $M_1 \neq M_2$, and (2) $\forall x \in \mathbf{U}$, $M_1(x) < M_2(x) \Rightarrow \exists y \in \mathbf{U}$, $[y > x$ and $M_1(y) > M_2(y)]$.

In plain words, the above definition says that for any element which occurs more in M_2 than in M_1, there is a greater element occurring more in M_1 than in M_2.

The above mathematical definition is however impractical as an algorithmic characterization of the multiset ordering. What is needed is an equivalent definition which can readily be translated into an efficient checking procedure. The following is yet another characterization that will be used as a construc-

tive method to establish whether or not a given multiset is greater than another. In words, the following proposition means that the lesser multiset M_2 can be obtained from M_1 by replacing some occurrences of elements of M_1 with arbitrarily many occurrences – possibly none – of lesser elements.

PROPOSITION 2.1: $M_1 \gg M_2$ *iff* (1) $M_1 - M_2 \neq \emptyset$; *and* (2) $(M_2 - M_1) - \{(x, (M_2 - M_1)(x)) \mid \exists y \in M_1 - M_2,\ y > x\} = \emptyset$ (*i.e., the multiset obtained from* $M_2 - M_1$ *after deletion of all occurrences of elements lesser than some element of* $M_1 - M_2$ *is empty*).

Proof: The following table sums up all possible cases (0 stands for non-empty):

Case	M_1^*		M_2	$M_1 - M_2$	$M_2 - M_1$	$M_1 \gg M_2$
1.	$\emptyset$		$\emptyset$	$\emptyset$	$\emptyset$	NO
2.	0		$\emptyset$	0	$\emptyset$	YES
3.	$\emptyset$		0	$\emptyset$	0	NO
4.	0	$=$	0	$\emptyset$	$\emptyset$	NO
5.	0	$\neq$	0	$\emptyset$	0	NO
6.	0	$\neq$	0	0	$\emptyset$	YES
7.	0	$\neq$	0	0	0	?

Let's refer to proposition 2.1 with the letter "*P*" and to definition 2.2 with the letter "*D*". Their respective clauses will then be referred to as *P*.1 and *P*.2 (resp., *D*.1 and *D*.2). Let's first prove the "if" direction; *i.e.*, that *P* implies *D*.

Cases 1, 3, 4, 5 are characterized by the fact that $M_1 - M_2 = \emptyset$. Therefore, if $M_1 - M_2 \neq \emptyset$, only cases 2, 6, and 7 are to be considered. Cases 2 and 6 are readily concluded. The only case remaining is 7. Let then $x \in \mathbf{U}$ such that $M_2(x) > M_1(x)$. By definition of the multiset difference operation, this is equivalent to $(M_2 - M_1)(x) > 0$. Hence, by *P*.2, $\exists y \in M_1 - M_2$ such that $y > x$. This immediately implies *D*.2.

Conversely, let's assume *D*. Now, suppose that $M_1 - M_2 = \emptyset$. The only case not contradicting *D*.1 is case 3. However in that case, by *D*.2, for any x in $M_1 - M_2$, there exists some y such that $y > x$ and $(M_1 - M_2)(y) > 0$; and this is a contradiction. Therefore, $M_1 - M_2 \neq \emptyset$. Next, suppose that *P*.2 does not hold; namely, let $x \in M_2 - M_1$ and $\forall y \in M_1 - M_2$, $y \not> x$. But, by *D*.2, there is a $y \in M_1 - M_2$ such that $y > x$; a contradiction.

Q.E.D.

Proposition 2.1 is thus proven to be an equivalent definition of the multiset extension of an ordering on a set.

COROLLARY: *The algorithm in figure* 2.1 *returns the value* **true** *iff its two multiset arguments* M_1 *and* M_2 *are such that* $M_1 \gg M_2$.

```
Function MultisetGreater(M1,M2: multiset of item): boolean;
    var M12,M21: multiset of item;
    X,Y: item;
    begin
    M12:=M1−M2; M21:=M2−M1;
    if M12=∅ then return(false)
    else
        begin
        while (M12≠∅) and (M21≠∅) do
            begin
            let X in 12;
            M12:=M12−{(X,M12(X))};
            M21:=M21−{(Y,M21(Y))|Greater(X,Y)}
            end;
        return(M21=∅)
        end
    end MultisetGreater;
```

Figure 2.1

3.0 TERM REWRITING SYSTEMS AND TERM ORDERING

In this section, some background and results on term-rewriting systems and term orderings are presented. The scope is strictly limited to the material relevant to the comprehension of the algorithm described in the following section. For a thorough introduction to the subject of term-rewriting systems, the reader is referred to [HOP80].

DEFINITION 3.1: Given a set **F** of function symbols and a set **V** of variable symbols, we define **T**(**F**, **V**), the set of (first-order) *terms* on **F** as follows:

(1) a variable is a term;

(2) any symbol in **F** is a term;

(3) if $f \in \mathbf{F}$ and $t_1, \ldots, t_n$ are terms, then so is $f(t_1, \ldots, t_n)$.

If $t = f(t_1, \ldots, t_n)$, f is called the *root* symbol of t, and t_i, $i = 1, \ldots, n$ are called the *arguments* of f in t. Note that the symbols in **F** have variable arity; that is, any function symbol may have 0 or any finite number of arguments. One writes f instead of $f()$ when f has no arguments.

In the sequel, the concept of multiterm rather than that of a term will be used. This is defined in the following inductive definition.

DEFINITION 3.2: Given sets of function and variable symbols **F** and **V** as above, the set of (first-order) *multiterms* **MT**(**F**, **V**) on **F** is the smallest set containing **V** and verifying the following property: if $f \in \mathbf{F}$, and M is a multiset of elements in **MT**(**F**, **V**), then $(f, M) \in \mathbf{MT}(\mathbf{F}, \mathbf{V})$.

To simplify the notation, we shall conventionally write f instead of $(f, \emptyset)$. An example of a multiterm is $M = (f, \{((g, \{a, b\}), 2), (a, 3)\})$. The multiterm concept will be useful in expressing the notion of equality of terms up to a

permutation of their subterms. Indeed, a multiterm can be seen as representing a class of terms sharing the same root symbol and the same multiset of subterms. For example, the terms $f(g(a, b), a, a, g(b, a), a)$ and $f(g(b, a), a, g(b, a), a, a)$ both belong to the same class represented by the multiterm M. Thus, any multiterm represents many possible terms. The following formally defines this idea. The mapping defined in definition 3.4 yields the multiterm representative (or image) of a given term.

DEFINITION 3.4: We define a mapping $\mathbf{M}$ from $\mathbf{T}(\mathbf{F}, \mathbf{V})$ to $\mathbf{MT}(\mathbf{F}, \mathbf{V})$ associating a multiterm on $\mathbf{F}$ to a term on $\mathbf{F}$ as follows:

$\mathbf{M}(t) = t$ if $t \in \mathbf{F} \cup \mathbf{V}$;

$\mathbf{M}(f(t_1, \ldots, t_n)) = (f, \{(s_1, k_1), \ldots, (s_m, k_m)\})$ where: $k_1 + \ldots + k_m = n$ and $\forall i \in \{1, \ldots, m\}$, $s_i = \mathbf{M}(t_j)$ for k_i indices $j \in \{1, \ldots, n\}$.

This apparently complicated definition means that for any term $t = f(t_1, \ldots, t_n)$, $\mathbf{M}(t) = (f, M)$, where M is the multiset of the multiterm images of the subterms $t_1, \ldots, t_n$. For example, if $t = f(g(a, b), f(a, c, a), g(b, a))$, then using our notational conventions, the above definition yields $\mathbf{M}(t) = (f, \{((g, \{a, b\}), 2), (f, \{(a, 2), c\})\})$.

It comes as an immediate observation that the mapping $\mathbf{M}$ is surjective; and hence, the following identity holds: $\mathbf{M}(\mathbf{T}(\mathbf{F}, \mathbf{V})) = \mathbf{MT}(\mathbf{F}, \mathbf{V})$. This allows us now to define the following relation $\cong$ on the terms, which can straightforwardly be checked to be an equivalence relation.

DEFINITION 3.5: $t_1 \cong t_2$ iff $\mathbf{M}(t_1) = \mathbf{M}(t_2)$, for any terms t_1 and t_2 in $\mathbf{T}(\mathbf{F}, \mathbf{V})$.

As a consequence, we obtain that the quotient set $\mathbf{T}(\mathbf{F}, \mathbf{V})/\cong$ is in bijection with the set $\mathbf{MT}(\mathbf{F}, \mathbf{V})$ of all multiterms on $\mathbf{F}$. This formalism enables us to deal only with multiterms as canonical representatives of terms in what follows.

A *term-rewriting system* is a finite set of pairs of terms. Such pairs are called *rules*, each rule (L, R) being such that all variables which occur in R also occur in L. Figure 3.1 exhibits a term-rewriting system whose rules rewrite (*cf.* definition 3.8) a quantifier-free logical formula into disjunctive normal form.

$$\begin{aligned}
\text{not}(\text{not}(u)) &\rightarrow u \\
\text{not}(\text{or}(u,v)) &\rightarrow \text{and}(\text{not}(u),\text{not}(v)) \\
\text{not}(\text{and}(u,v)) &\rightarrow \text{or}(\text{not}(u),\text{not}(v)) \\
\text{and}(u,\text{or}(v,w)) &\rightarrow \text{or}(\text{and}(u,v),\text{and}(u,w)) \\
\text{and}(\text{or}(u,v),w) &\rightarrow \text{or}(\text{and}(u,w),\text{and}(v,w))
\end{aligned}$$

Figure 3.1

The rules of a first-order term-rewriting system are representative schemata of infinitely many rules by virtue of their variables. The concept of substitution of variables is thus necessary to express instantiation.

DEFINITION 3.6: A *substitution* **s** is an application from **V** to **T**(**F**, **V**) which is the identity on **V** almost everywhere except for a finite set of variables $\{x_1, \ldots, x_n\}$.

A substitution **s** is easily extended to an application $\hat{\mathbf{s}}$ from **T**(**F**, **V**) to **T**(**F**, **V**) as follows:

$\hat{\mathbf{s}}(t) = \mathbf{s}(t)$ if $t \in \mathbf{V}$;

$\hat{\mathbf{s}}(t) = t$ if $t \in \mathbf{F}$;

$\hat{\mathbf{s}}(f(t_1, \ldots, t_n)) = f(\hat{\mathbf{s}}(t_1), \ldots, \hat{\mathbf{s}}(t_n))$ otherwise.

DEFINITION 3.7: A rule (L, R) is said to be a *reduction* if there is a strict partial ordering $>$ on **T**(**F**, **V**) such that $\hat{\mathbf{s}}(L) > \hat{\mathbf{s}}(R)$, for all substitutions **s**.

As shall be seen later, the rules of the rewriting system in figure 3.1 are reductions with respect to a recursive path ordering (*cf.* definition 3.11).

The structure of a term is partially altered by rule application. Hence, it is of great convenience to have a precise scheme for specifying how and what particular part of it is to change. For this, we use a simple formalism due to S. Gorn [GOR65]. Given a term t, we define a *term address* in t denoting a specific place in t as a finite word of positive integers such that (1) the empty word **e** is a term address of t; and (2) if $t = f(t_1, \ldots, t_n)$, iu is a term address of t, where u is a term address of t_i, for $i = 1, \ldots, n$. The subterm of t at address u (noted t_u) is t if $u = \mathbf{e}$, or is $(t_i)_v$ if $u = iv$ and $t = f(\ldots, t_i, \ldots)$. The term denoted by $t[u \leftarrow t']$ is the term obtained from t by replacing t_u with t'. A term t is said to *occur* in another term t' at address u, if u is a term address of t' and $t'_u = t$.

For any set of rules, one can define a binary relation $\rightarrow$ on terms expressing rule application.

DEFINITION 3.8: Given A term-rewriting system $\{(L_i, R_i)\}_{i=1}^n$ on **T**(**F**, **V**), a term t_1 *rewrites* to a term t_2 (noted as $t_1 \rightarrow t_2$) if there is an index $i \in \{1, \ldots, n\}$ and a substitution **s** such that $\hat{\mathbf{s}}(L_i)$ occurs in t_1 at address u, and $t_2 = \hat{\mathbf{s}}(t_1[u \leftarrow R_i])$.

The transitive closure of $\rightarrow$ is written $\overset{+}{\rightarrow}$. This allows us to define the concept of a terminating set of rewriting rules.

DEFINITION 3.9: A *well-founded* partial ordering $>$ on a set **S** is one such that there is no infinitely descending chain $s_1 > s_2 > \ldots > s_n > \ldots$ in **S**.

DEFINITION 3.10: A term-rewriting system is *finite terminating* (or *noetherian*) iff the relation $\overset{+}{\rightarrow}$ is well-founded.

In effect, this means that there may not exist an infinite rewriting sequence $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow \ldots$ of terms.

DEFINITION 3.11 [DER82.*a*]: A partial ordering $>$ on $\mathbf{T}(\mathbf{F}, \mathbf{V})$ is a *simplification ordering* if it has the following properties for any terms and function symbols:

(1) If $t_1 > t_2$ then $f(\ldots, t_1, \ldots) > f(\ldots, t_2, \ldots)$;

(2) $f(\ldots, t, \ldots) > t$;

(3) $f(\ldots, t_{i-1}, t_i, t_{i+1}, \ldots) > f(\ldots, t_{i-1}, t_{i+1}, \ldots)$.

These three properties are respectively called the (1) *monotonicity*, (2) *subterm*, and (3) *deletion* property.

THEOREM 3.1 (Dershowitz' First Termination Theorem [DER82.*a*]): *A term-rewriting system* $\{(L_i, R_i)\}_{i=1}^{n}$ *on* $\mathbf{T}(\mathbf{F}, \mathbf{V})$ *is finite terminating if there exists a simplification ordering* $>$ *on* $\mathbf{T}(\mathbf{F}, \mathbf{V})$ *such that* $\hat{\mathbf{s}}(L_i) > \hat{\mathbf{s}}(R_i)$, $\forall i \in \{1, \ldots, n\}$, *for all substitutions* **s**.

We have finally arrived at the point where we can define the concept of recursive path ordering on the set of multiterms $\mathbf{MT}(\mathbf{F}, \mathbf{V})$.

DEFINITION 3.12: Given a strict partial ordering $>$ on the set of function symbols **F**, the *recursive path ordering* (RPO) $\overset{*}{>}$ induced by $>$ on the set of multiterms $\mathbf{MT}(\mathbf{F}, \mathbf{V})$ is defined recursively as: $(f, S) \overset{*}{>} (g, T)$ iff one of the following holds:

(1) $f = g$ and $S \overset{**}{\gg} T$;

(2) $f > g$ and $\{(f, S)\} \overset{**}{\gg} T$;

(3) $f \not> g$ and $S \overset{**}{\underline{\gg}} \{(g, T)\}$.

In this definition, variables are treated as incomparable constant symbols. Recall from section 2 that given an ordering symbol $>$, $\gg$ is its multiset ordering extension symbol. Also, $\underline{\gg}$ is its reflexive closure symbol.

The rules in figure 3.1 form a set of RPO-reductions given that *not* $>$ *and* $>$ *or*. The rules in figure 3.2 are also RPO-reductions if *reverse* $>$ *append* $>$ *cons* $>$ *false* and *reverse* $>$ *nil* $>$ *true*.

$$\begin{aligned}
\text{head}(\text{cons}(u,v)) &\to u \\
\text{tail}(\text{cons}(u,v)) &\to v \\
\text{empty} &\to \text{true} \\
\text{empty}(\text{cons}(u,v)) &\to \text{false} \\
\text{append}(\text{nil},u) &\to u \\
\text{append}(\text{cons}(u,v),w) &\to \text{cons}(u,\text{append}(v,w)) \\
\text{reverse}(\text{nil}) &\to \text{nil} \\
\text{reverse}(\text{cons}(u,v)) &\to \text{append}(\text{reverse}(v),\text{cons}(u,\text{nil}))
\end{aligned}$$

Figure 3.2

The next and last two theorems of this section explain why a RPO is at all interesting in proving termination of a term-rewriting system.

THEOREM 3.2 ([DER82.*a*]): *The ordering* $\mathbf{M}^{-1}(\overset{*}{>})$ *on* $\mathbf{T}(\mathbf{F},\mathbf{V})$ *is a simplification ordering.*

THEOREM 3.3 ([DER82.*a*]): *The ordering* $\mathbf{M}^{-1}(\overset{*}{>})$ *on* $\mathbf{T}(\mathbf{F},\mathbf{V})$ *is well-founded iff the underlying ordering* $>$ *on* $\mathbf{F}$ *is well-founded.*

Hence, finding a well-founded partial ordering $>$ on the function symbols which induces a RPO with respect to which all rules (L, R) in a set are such that $\mathbf{M}(\hat{\mathbf{s}}(L)) \overset{*}{>} \mathbf{M}(\hat{\mathbf{s}}(R))$ for all substitutions $\mathbf{s}$, is a sufficient way of proving that the set of rules is noetherian. This is the problem that the algorithm presented in the next section solves.

4.0 ALGORITHM FOR FINDING A MINIMAL RPO

In the previous section, the definition of the recursive path ordering is based on the *a priori* knowledge of a partial ordering on the function symbols. Hence, a sufficient condition for termination of a term-rewriting system would be at hand if such a well-founded ordering on the function symbols could somehow be obtained. It would even be of greater convenience if a method for obtaining it could be fully automated. Thence, given a set of rules, one could simply run a program which would make that sufficient condition hold if one can be made to hold at all.

A similar method, unknown to the author when this work was done, is proposed in [LES83.*a*] and has been integrated in the REVE system [LES83.*b*], based on an extension of the recursive path ordering: the *recursive decomposition ordering* [LES82, JLR83]. Although it is argued that the recursive decomposition ordering is more general (but see the appendix), it is more "tricky" to define. However, no formal comparison has been made between that method and the method presented in this paper.

In what follows, an algorithm is described which finds a minimal strict partial ordering on the function symbols of a set of rules for which all the rules are RPO-reductions. The method simply verifies definition 3.12 with no prior knowledge of any ordering on the function symbols, making the minimal necessary assumptions as it follows the induction in order to make the definition hold for all possible substitutions of the variables. Figures 4.1-4.4 are a pseudo-Pascal sketch of the method. In the following section, a complete and detailed Prolog implementation is shown.

The function Find_RPO (*cf. fig.* 4.1) takes three arguments: a set of rewrite rules, a partially ordered set (poset) of function symbols, and an output parameter which is also a poset of function symbols. The input poset is a background poset of what is already known to be ordered. The output poset is thus a consistent augmentation of the former poset. The function returns the value **true** if such a consistent poset is found.

The body of Find_RPO is essentially a loop of recursive calls to itself based on the following intuitive argument. It computes a minimal poset consistent with the given background poset such that the first rule in the set has its left-hand side strictly greater than its right-hand side. Then, with this poset as background, it will try to find a minimal poset for the rest of the rules. In case no poset can be found consistent with this background, it is rejected and remembered as such in Deja_Vu, a set of posets.

The function RPO_Greater (*cf. fig.* 4.2) compares two multiterms given a background poset, a set of previously rejected posets, by a "lazy" construction of an output poset following the minimal requirements that the definition of the Recursive Path Ordering hold. It works as follows.

If the two root symbols are the same, then there is no alternative but to check whether the left multiset of submultiterms is greater than the right one according to the multiset extension of an RPO consistent with the context. This also clearly covers the case dealing with two identical variables since both occurrences are always replaced with the same terms in any rewriting.

If the context shows that the left root symbol f is prevented from being greater than the right root symbol g—*i.e.*, that Frozen (f, g, Past) holds, where Past is the background poset—or if one of them is a variable, then all there is to do is verify whether the case "$f \triangleright g$" of the RPO definition yields a consistent poset. The case where one of the symbols is a variable is handled here by always following the "$f \triangleright g$" clause of the definition. This is because we want the definition to hold for all substitutions of the variables. It is clear that making any presumption on the order of two different variables, or a variable and a symbol would not be consistent for *all* substitutions.

If the context shows that $f > g$, then the "$f > g$" case of the definition is similarly checked.

Only if none of the above applies is it made any assumption about whether the root symbols are related or not. The first alternative tried is to assume that f is not greater than g. The function Freeze does that. It takes as arguments f, g, and a poset, and returns the poset augmented with the constraint that $f > g$ not be allowed to hold later on down the search. In the case where this assumption leads nowhere, the opposite (*i.e.*, $f > g$) is attempted, provided it is consistent. The function Consistent (f, g, Past) will return the value **false** iff there exist some x, y in the poset Past such that $x \geqq f$ and $g \geqq y$ and Frozen (x, y, Past).

If in the end the assumptions fail, RPO_Greater also fails.

The function Multi_RPO_Greater (*cf. fig.* 4.3) checks whether a multiset of multiterms is greater than another in a given context, using a multiset extension of RPO_Greater. The algorithm used is a slight fix of the Multiset_Greater function in section 2. Indeed, the difference is in the fact that RPO_Greater is conceived to possibly have side-effects. Namely, it may augment the background poset to accommodate for success. Hence, the need for the Prune function (*cf. fig.* 4.4) which simply computes the result of pruning a multiset of its elements which are less than a given multiterm, and propagates any background expansion along the way.

```
Function Find_RPO
    (
    Rule_Set: set of rules;
    Background: poset;
    var Order: poset
    ): boolean;
    var Deja_Vu: set of poset;
        So_Far: poset;
        Done: boolean;
    begin
    if Rule_Set = ∅ then return(Background)
    else
        begin
        let L → R in Rules;
        Rule _Set: = Rule_Set − { L → R };
        Done: = false;
        Deja_Vu: = ∅;
        while (not Done) and RPO_Greater(L,R,Background,Deja_Vu,So_Far) do
            begin
            Deja_Vu: = Deja_Vu ∪ { So_Far };
            Done: = Find_RPO(Rule_Set,So_Far,Order)
            end;
        return(Done)
        end
    end Find_RPO;
```

Figure 4.1

```
Function RPO_Greater
    (
    L,R: multiterm;
    Past: poset;
    Deja_Vu: set of posets;
    var Order: poset
    ): boolean;
    begin
    let L=(f,S), R=(g,T);
    if f=g
        then return(Multi_RPO_Greater(S,T,Past,Deja_Vu,Order))
    else
    if Greater(f,g,Past)
        then return(Multi_RPO_Greater({L},T,Past,Deja_Vu,Order))
    else
    if Variable(f) or Variable(g) or Frozen(f,g,Past)
        then return(Multi_RPO_Greater_or_Equal(S,{R},Past,Deja_Vu,Order))
    else
    if Multi_RPO_Greater_or_Equal(S,{R),Freeze(f,g,Past),Deja_Vu,Order)
        then return(true)
    else
    if Consistent(f,g,Past)
        then return(Multi_RPO_Greater({L},T,Freeze(g,f,Past ∪ {f>g}), Deja_Vu,Order))
    else return(false)
    end RPO_Greater;
```

Figure 4.2

```
Function Multi_RPO_Greater
    (
    M1,M2: multiset of multiterm;
    Past: poset;
    Deja_Vu: set of poset;
    var Order: poset
    ): boolean;
    var M12,M21: multiset of multiterm;
        X: multiterm;
    begin
    M12:=M1−M2; M21:=M2−M1;
    if M12=∅ then return(false)
    else
        begin
        while (M12≠∅) and (M21≠∅) do
            begin
            let X in M12;
            M12:=M12−{(X,M12(X))};
            M21:=Prune(M21,X,Past,Deja_Vu,Order);
            Past:=Order
            end;
        return((M21=∅) and not(Order in Deja_Vu))
        end
    end Multi_RPO_Greater;
```

Figure 4.3

```
Function Prune
    (
    M: multiset of multiterm;
    X: multiterm;
    Past: poset;
    Deja_Vu: set of poset;
    var Order: poset
    ): multiset of multiterm;
    var Y: multiterm;
        N: multiset of multiterm;
    begin
    N:=M;
    while N≠∅ do
        begin
        let Y in N;
        N:=N−{(Y,N(Y))};
        if RPO_Greater(X,Y,Past,Deja_Vu,Order) then
            begin
            M:=M−{(Y,M(Y))};
            Past:=Order
            end
        end;
    return(M)
    end Prune;
```

Figure 4.4

Claim: *The algorithm described in the foregoing figures* 4.1-4.4 *is correct. That is,* Find_RPO *always terminates, and it returns the value* **true** *if and only if the poset Order induces a minimal Recursive Path Ordering such that each rule in the given set is a strict reduction.*

Informal Justification: The justification is better presented in four parts: (1) termination; (2) in case of success, the ordering defined by Order induces a RPO, and (3) Order is minimal; finally, (4) if the value **false** is returned, no RPO exists which makes the rules be strict reductions.

Termination of Find_RPO follows from the fact that there are finitely many posets on a finite set, together with the observation that RPO_Greater calls Multi_RPO_Greater each time with multisets of strictly shallower multiterms. Indeed, in each call to Multi_RPO_Greater, at least one of the two multiterm argument's depth is strictly decreasing.

If the value returned by Find_RPO is **true**, as the poset Order is constructed in RPO_Greater conformingly to definition 3.12, it clearly defines an ordering on the function symbols inducing a RPO which verifies the reduction contraints.

It is also a built-in feature that the poset returned as Order defines a minimal ordering; minimal in the sense that taking any pair out of it would make it be not consistent with the set of given rules. This is a result of

making a least assumption at any point about whether two function symbols are related. The algorithm RPO_Greater assumes that two symbols are related only if everything else has failed.

An interesting observation is that if the two last "if" statements were swapped, minimality would not be guaranteed; however, for most applications, the time taken by the algorithm to succeed would be quite reduced since it would first try and make the assumption that two function symbols are related. There seems to be a trade-off between minimately and time-efficiency. One could argue for the latter, since minimately is nothing of great interest as far as proving termination is concerned.

I have not as yet been able to find a convincing argument to defend the fourth point. It is made more difficult than the others by what is really happening in the function Prune. Indeed, this function removes from a multiset all multiterms RPO-inferior to a given multiterm. However it does this sequentially, and any decisions made on the way are definite and irretractable. Hence, in order to prove that this procedure does not miss on any potential solution, one is required to show some kind of a sufficient condition lemma arguing that one need never backtrack at that point. I suspect that to be correct and must be elucidated if the full correctness proof is to be given. This will done eventually in a forthcoming paper.

However, for all purposes here concerned, the Prolog implementation in the next section does not make the assumption implicit in Prune. Namely, it allows for alternative pruning. Incidentally, actual backtracking at that point after initial failure has never been observed to succeed in all the examples tried. That tends to give empirical credit to my suspicion above.

5.0 THE PROLOG IMPLEMENTATION

This section is the detailed description of the Prolog program implementation of the algorithm sketched in the previous section. It is interesting in many respects, not the least of which being the implicit mathematical induction proof of correctness that is "built-in". Indeed, a Prolog program can be loosely construed, by its very syntactic form, as "self-proving" its own correctness. One may read it so the inductive definition of each predicate constitutes a proof by case induction. Each definition is typically a base case followed by all possible structural induction cases. Of course, this does not claim to be a formal argument for the correctness of the algorithm. However a correctness proof would bear very close appearence to the following Prolog program.

setOfReductions (Rules) succeeds if there can be found a recursive path ordering on the terms for which all pairs (left, right) in a set of rewrite rules are strict reductions—*i. e.*, left > right:

```
setOfReductions([]).
setOfReductions([[Left,Right]|Rules]): -
        convertRule(Left,Right,Mleft,Mright),
        rpoGreater(Mleft,Mright),
        setOfReductions(Rules).
```

convertRule (Left,Right,Mleft,Mright) converts each side of a rule to their respective multiterm images:

```
convertRule(Left,Right,Mleft,Mright): -
        convert(Left,L),
        multiTerm(L,Mleft),
        convert(Right,R),
        multiTerm(R,Mright).
```

convert(Term, List) converts a term in prefix parenthesized form into its equivalent list form. *e. g.*:

$$\text{Term} = f\,(g\,(a,\,b),\,f\,(a,\,c,\,a),\,g\,(b,\,a)),$$

$$\text{List} = [\,f,\,[g,\,[a],\,[b]],\,[\,f,\,[a],\,[c],\,[a]],\,[g,\,[b],\,[a]]],$$

```
convert(Term,[F|SubLists]): -
        Term = . . [F|SubTerms],
        map2(convert,SubTerms,SubLists).
```

multiTerm(T, MT) succeeds if the term T is of the form $[\,f,\,t_1,\,\ldots,\,t_n]$—constants and variables are represented as one element lists: $[a]$—*and* MT is the multiterm $[\,f \mid M]$, where M is the multiset of multiterms of the form $[[M_1,\,N_1],\,\ldots,\,[M_m,\,N_m]]$, $m \leqq n$, where M_i is a multiterm and $N_i \geqq 1$ is the multiplicity of M_i in M. Recall that two terms are considered equal if they are the same up to a permutation of their subterms. For example:

$$T = [\,f,\,[g,\,[a],\,[b]],\,[g,\,[b],\,[a]],\,[\,f,\,[c],\,[a],\,[c]]],$$
$$MT = [\,f,\,[[g,\,[[a],\,1],\,[[b],\,1]],\,2],\,[[\,f,\,[[c],\,2],\,[[a],\,1]],\,1]],$$

```
multiTerm([F|T],[F|M]): - multiSetOfMultiTerms(T,M).
```

multiSetOfMultiTerms(Past,TermSet,MtermMset) means that MtermMset is the multiset of multiterm images corresponding to the set of terms TermSet. Past is a book-keeping multiset of multiterms of already seen elements, and updated on the way. This is what MtermMset eventually becomes:

```
multiSetOfMultiTerms([],[]): -!.
multiSetOfMultiTerms(Tset,MtMset): -
        multiSetOfMultiTerms([],Tset,MtMset).
multiSetOfMultiTerms(Past,[],Past).
multiSetOfMultiTerms(Past,[Term|Tset],MtMset): -
        multiTerm(Term,Mterm),
        deja_vu(Mterm,Past,[[Mterm,N1]|OtherPast]),
        N is N1+1,
        multiSetOfMultiTerms([[Mterm,N]|OtherPast],Tset,MtMset),!.
```

deja_vu(Mterm,Past,Carry,MtMset) succeeds if the multiterm Mterm has already been seen in Past and MtMset is the multiset of multiterms [[Mterm,N]|M] where M is (Past-[Mterm,N]) $\cup$ Carry. This predicate has nothing to do with the set Deja_Vu in the pseudo-Pascal description of the previous section:

```
deja_vu(Mterm,Past,MtMset): – deja_vu(Mterm,Past,[],MtMset).
deja_vu(Mterm,[],Carry,[[Mterm,0]|Carry]): –!.
deja_vu(Mterm1,[[Mterm2,N]|Past],Carry,[[Mterm1,N]|Rest]): –
        sameMultiTerm(Mterm1,Mterm2),append(Past,Carry,Rest),!.
deja_vu(Mterm,[Msingle|Past],Carry,MtMset): –
        deja_vu(Mterm,Past,[Msingle|Carry],MtMset).
```

sameMultiTerm($MT1,MT2$) is true if $MT1$ and $MT2$ are the same multiterms. For example:

$$MT1 = [f, [[g, [[a], 1], [[b], 1]], 2], [[f, [[c], 2], [[a], 1]], 1]]$$

$$MT2 = [f, [[f, [[a], 1], [[c], 2]], 1], [[g, [[b], 1], [[a], 1]], 2]]$$

```
sameMultiTerm([],[]).
sameMultiTerm([F|T1],[F|T2]): – sameMultiSetOfMultiTerms(T1,T2).
sameMultiSetOfMultiTerms(MT1,MT2): – sameSetOfMultiTerms(MT1,MT2,[]).
sameSetOfMultiTerms([],[],[]).
sameSetOfMultiTerms([H1,N]|T1],[[H2,N]|T2],Carry): –
        sameMultiTerm(H1,H2),
        append(Carry,T2,T3),
        sameMultiSetOfMultiTerms(T1,T3),!.
sameSetOfMultiTerms(T1,[H2|2],Carry): –
        sameSetOfMultiTerms(T1,T2,[H2|Carry]).
```

rpoGreater($M1$, $M2$) succeeds if the multiterms $M1$ and $M2$ are such that $M1$ is greater than $M2$ according to a "lazy" recursive path ordering that is consistent with the Rules left yet to be verified as reductions. The RPO is incrementally checked on the way. What is really built on the way is the underlying strict partial ordering on the function symbols "o". That is, $o(x, y)$ is true iff x is greater than y:

```
rpoGreater([F|M1],[F|M2]): –
        !,multi_rpoGreater(M1,M2).
rpoGreater([F|M1],[G|M2]): –
        o(F,G),!,
        multi_rpoGreater([[[F|M1],1]],M2).
rpoGreater([F|M1],[G|M2]): –
        explore(F,G,M1,M2).
```

The ordering **o** must be transitive:

```
o(X,Y): = var(Y),!,fail.
o(X,Y): – o(X,Z),o(Z,Y).
```

explore($F,G,M1,M2$) attempts to make rpoGreater succeed by constraining F and G to be either unrelated or $o(G, F)$. **frozen**(f,g) means that $o(f, g)$ cannot hold:

```
explore(F,G,M1,M2):-
        (variable(F);variable(G);frozen(F,G)),
        !,multi_rpoGreaterOrEqual(M1,[[[G|M2],1]]).
explore(F,G,M1,M2):-
        freeze(F,G),
        multi_rpoGreaterOrEqual(M1,[[G|M2],1]]).
explore(F,G,M1,M2):-
        unfreeze(F,G),
        attempt(F,G),
        multi_rpoGreater([[[F|M1],1]],M2).
explore(F,G,_,_):-reject(F,G),!,fail.
variable(X):-nonvar(X),declaredVariable(X).
```

freeze(F,G) forbids that $o(F, G)$ be asserted further down in the search. That is, success of all further subgoals is conditioned by the constraint that not ($o(F, G)$):

```
freeze(F,G):-
        asserta(frozen(F,G)),
        tr(forbidding,(G<F)).
```

unfreeze(F,G) relaxes the above constraint:

```
unfreeze(F,G):-retract(frozen(F,G)),tr(allowing,(G<F)).
```

attempt(F,G) hypothesizes that $F>G$:

```
attempt(F,G):-
        not(inconsistent(F,G)),
        asserta(o(F,G)),
        tr(attempting,(F>G)),
        freeze(G,F).
```

inconsistent(F,G) succeeds if there is some X greater or equal to F, and some Y smaller or equal to G, such that X and Y are frozen. That is, if F cannot be made greater than G. **o_eq** is the reflexive closure of o:

```
inconsistent(F,G):-o_eq(X,F),o_eq(G,Y),frozen(X,Y),!.
o_eq(X,X).
o_eq(X,Y):-o(X,Y).
```

reject(F,G) establishes $o(F,G)$ to be no longer a hypothesis that holds:

```
reject(F,G):-
        retract(o(F,G)),
        tr(rejecting,(F>G)),
        retract(frozen(G,F)),
        tr(allowing,(F<G)).
```

multi_rpoGreaterOrEqual($M1,M2$) is the reflexive closure of the following predicate:

```
multi_rpoGreaterOrEqual(M1,M2):-sameMultiSetOfMultiTerms(M1,M2),!.
multi_rpoGreaterOrEqual(M1,M2):-multi_rpoGreater(M1,M2).
```

multi_rpoGreater($MT1, MT2$) succeeds if the multisets of multiterms $MT1$ and $MT2$ are such that $MT1 \overset{*}{>} MT2$ according to the multiset extension of the rpoGreater ordering:

```
multi_rpoGreater([],M2):-!,fail.
multi_rpoGreater(M1,[]):-!.
multi_rpoGreater(M1,M2):-
        split(M1,M2,M1_M2,M2_M1),
        weigh(M1_M2,M2_M1).
```

split($M1, M2, M1\text{-}M2, M2\text{-}M1$) succeeds by simultaneously building the symmetrical difference multisets:

```
split([],M,[],M).
split(M,[],M,[]).
split([[X,N]|M1],M2,D1,D2):-
        in(X,M2,[[X,K]|M21]),!,
        D is N-K,
        deal(X,D,M1,M21,D1,D2).
split([[X,N]|M1],M2,D1,D2):-deal(X,N,M1,M2,D1,D2).
```

in($X, M1, M2$) succeeds if a multiterm equal to X is in $M1$ — in which case $M2$ is $[[X,N]|M2 - \{[X,N]\}]$:

```
in(Elt,M1,M2):-in([],Elt,M1,M2).
in(_,_,[],_):-!,fail.
in(Carry,Elt,[[X,N]|M1],[[Elt,N]|M2]):-
        sameMultiTerm(Elt,X),!,
        append(Carry,M1,M2).
in(Carry,Elt,[[X,N]|M1],M2):-in([[X,N]|Carry],Elt,M1,M2).
```

deal($X, N, M1, M2, D1, D2$) distributes X in $D1$ if $N<0$, or in $D2$ if $N>0$, or in neither if $N=0$. $D1$ and $D2$ are then further "split" from $M1$ and $M2$:

```
deal(_,0,M1,M2,D1,D2):-split(M1,M2,D1,D2),!.
deal(X,D,M1,M2,[[X,D]|D1],D2):-D>0,!,split(M1,M2,D1,D2).
deal(X,D,M1,M2,D1,[[X,MD]|D2]):-MD is -D,!,split(M1,M2,D1,D2).
```

weigh($M1, M2$) tries to find a consistent RPO to make $M1$ and $M2$ such that $M2$ is a partition of subsets, each of which is majored by an element of $M1$:

```
weigh([],M):-!,fail.
weigh(M,[]):-!.
weigh(M1,M2):-major(M1,M2).
```

major($M1, M2$,Rem) will succeed if a RPO can be found consistent such that $\text{Rem} = M2 - \{Y | \text{rpoGreater}(X,Y)\}$ and major $(M1 - \{X\}, \text{Rem})$ succeeds, for each element X of $M1$:

```
major(_,[]).
major([],_):-!,fail.
major([[X,_]|M1],M2):-
        minor(X,M2,Rem),
        major(M1,Rem).
```

Minor(X,M,Rem) iff $\text{Rem} = M - \{Y | \text{rpoGreater}(X,Y)\}$:

```
minor(_,[],[]).
minor(X,[[Y,_]|M],Rem):-
        rpoGreater(X,Y),
        minor(X,M,Rem).
minor(X,[Y|M],[Y|Rem]):-minor(X,M,Rem).
```

tr(Trace,Info) is a tracing convenience...

```
tr(Trace,Info):-write(Trace),write(':'),tab,write(Info),nl,!.
```

6.0 CONCLUSION

This paper has presented an automatic procedure which infers a Recursive Path Ordering making a set of rewrite rules be a set of strict reductions, if one exists. By Dershowitz's results, this provides an automatic termination proof procedure. Indeed, these results stipulate that a well-founded RPO for which each rule is a reduction guarantees termination of a set of rules; moreover, a RPO is well-founded if and only if the underlying strict partial ordering on the function symbols is too. If the set of function symbols in a first-order term-rewriting system is finite, it is sufficient to find the underlying ordering to prove termination.

Using this algorithm, the Knuth-Bendix completion method can be run without imposing that a term ordering be *a priori* provided. The construction exposed in this paper gives the possibility to automate rule orientation. This is a further research step that I intend to take.

APPENDIX

This appendix contains sample runs of the Prolog program.

The set of rules **dnf1** is the same as the one in figure 3.1. This run was done with a trace enable visualization of the search path:

```
?-printRules(dnf1).

not(not(u))          →u
not(or(u,v))         →and(not(u),not(v))
not(and(u,v))        →or(not(u),not(v))
and(u,or(v,w))       →or(and(u,v),and(u,w))
and(or(u,v),w)       →or(and(u,w),and(v,w))
yes
?-try(dnf1).

forbidding:    and < not
forbidding:    and < or
allowing:      and < or
attempting:    or > and
forbidding:    or < and
forbidding:    not < or
allowing:      not < or
attempting:    or > not
forbidding:    or < not
```

```
rejecting:      or > not
allowing:       or < not
forbidding:     not < or
allowing:       not < or
attempting:     or > not
forbidding:     or < not
rejecting:      or > not
allowing:       or < not
rejecting:      or > and
allowing:       or < and
allowing:       and < not
attempting:     not > and
forbidding:     not < and
forbidding:     or < not
forbidding:     or < and
allowing:       or < and
allowing:       or < not
attempting:     not > or
forbidding:     not < or
forbidding:     or < and
allowing:       or < and
attempting:     and > or
forbidding:     and < or
and > or
not > and
yes
```

The following runs are done with disabled tracing. The rules **dnf2** also compute a disjunctive normal form as **dnf1**. However, the partial ordering found by the algorithm is different:

```
?- printRules(dnf2).

not(not(u))      →u
not(or(u,v))     →and(and(not(u),not(v)),and(not(u),not(v)))
not(and(u,v))    →or(or(not(u),not(v)),or(not(u),not(v)))
and(u,u)         →u
or(u,u)          →u
yes

?- try(dnf2).
not > or
not > and
yes
```

The rewriting system **dnf3** also computes disjunctive normal forms; however, it is not finite terminating (*see* [DER82. *a*]). Note that the algorithm terminates with a failure:

```
?- printRules(dnf3).

not(not(u))      →u
not(or(u,v))     →and(not(not(not(u))),not(not(not(v))))
not(and(u,v))    →or(not(not(not(u))),not(not(not(v))))
and(u,or(v,w))   →or(and(u,v,and(u,w))
yes

?- try(dnf3).
no
```

Assoc is an example of a terminating system which does not admit any recursive path ordering (*see* [DER82. *a*]):

```
?-printRules(assoc).
and(and(u,v),w)→and(u,and(v,w))
yes

?-try(assoc).
no
```

The following two examples are taken from [LES83. *a*] where they are used to show that the recursive decomposition ordering is stronger that the recursive path ordering. The author believes that this is not a real point since what Pierre Lescanne shows is that for a given *fixed* precedence ordering on the function symbols, a RPO exists whenever a RDO exists, but not conversely. He uses the two following examples to illustrate this. However, it is clear, as shown here, that in each case, one can find a RPO to prove termination. Considering the comparatively simpler definition of RPO, the point is moot:

```
?-printRules(lescanne1).
or(not(u),not(v))        →not(and(u,v))
yes

?-try(lescanne1).
or>and
or>not
yes

?-printRules(lescanne2).
h(star(h(u),u))→star(u,h(star(u,u)))
yes

?-try(lescanne2).
h>star
yes
```

The following example **bool** is a The following example **bool** is a set of canonical axioms for a boolean ring (*see* [DER82. *b*]). It is interesting to point out that the algorithm finds an ordering strictly contained in the one given by Dershowitz to prove termination (*see* [DER82. *b*], p. 11):

```
?-printRules(bool).
not(u)          →xor(u,true)
or(u,v)         →xor(and(u,v),xor(u,v))
implies(u,v)    →xor(and(u,v),xor(utrue))
and(u,true)     →u
and(u,false)    →false
```

```
and(u,u)          →u
xor(u,false)      →u
xor(u,u)          →false
and(xor(u,v),w)   →xor(and(u,w),and(v,w))

yes

?-try(bool).

and > xor
xor > false
implies > and
implies > true
or > and
not > true
not > xor

yes
```

The next set of rules **diff** does symbolic differentiation (*see* [DER82. *b*]). Note that the ordering inferred is different than the one given by Dershowitz in [DER82. *b*] (p. 8); however, both orderings are minimal:

```
?-printRules(diff).

d(X)              →1
d(a)              →0
d(plus(u,v))      →plus(d(u),d(v))
d(minus(u,v))     →minus(d(u),d(v))
d(minus(u))       →minus(d(u))
d(times(u,v))     →plus(times(v,d(u)),times(u,d(v)))
d(over(u,v))      →minus(over(d(u),v),times(u,over(d(v),times(v,v))))
d(log(u))         →over(d(u),u)
d(exp(u,v))       →plus(times(v,exp(u,minus(v,1)),d(u)),times(exp(u,v),log(u),d(u)))

yes

?-try(diff).

d > log
d > exp
d > over
d > times
d > minus
d > plus
a > 0
d > 1

yes
```

The last example **list** is the one also shown in figure 3.2 before. It defines basic axioms for list manipulations.

```
?-printRules(list).

head(cons(u,v))       →u
tail(cons(u,v))       →v
empty(nil)            →true
empty(cons(u,v))      →false
append(u,nil)         →u
append(cons(u,v,w)    →cons(u,append(v,w))
reverse(nil)          →nil
reverse(cons(u,v))    →append(reverse(v),cons(u,nil))
```

```
yes

?-try(list).

reverse > append
reverse > nil
append > cons
cons > false
nil > true
```

REFERENCES

[ADJ78] J. A. GOGUEN, J. W. THATCHER and E. G. WAGNER, *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*, Current Trends in Programming Methodology, in R. T. YEH Ed., V. 4, Prentice Hall, 1978.

[DER82] (*a*) N. DERSHOWITZ, *Orderings for Term-Rewriting Systems*, Theor. Comp. Sc., Vol. 17, (3), pp. 279-301;
(*b*) N. DERSHOWITZ, *Computing with Rewrite Systems*, Technical Paper Draft, Bar-Ilan University, August 1982.

[DRM79] N. DERSHOWITZ and Z. MANNA, *Proving Termination with Multiset Ordering*, Communications of the A.C.M., Vol. 22, (8), 1979, pp. 465-476.

[GOT79] J. A. GOGUEN and J. J. TARDO, *An Introduction to OBJ: a Language for Writing and Testing Formal Algebraic Program Specifications*, Proceedings of the I.E.E.E. Conference on Specifications of Reliable Software, Cambridge, MA, 1979, pp. 170-189.

[GOR65] S. GORN, *Explicit Definitions and linguistic Dominoes*, in Systems and Computer Science, J. HART, S. TAKASU Eds., University of Toronto Press, 1965.

[HOD82] C. M. HOFFMAN and M. O'DONNELL, *Programming with Equations*, A.C.M. Transactions on Programming Languages and Systems, Vol. 4, (1), 1982, pp. 83-112.

[HOP80] G. HUET and D. C. OPPEN, *Equations and Rewrite Rules*, in Formal Language Theory, Perspective and Open Problems, R. BOOK Ed., Academic Press, 1980, pp. 349-393.

[HUE81] G. HUET, *A Complete Proof of Correctness of the Knuth-Bendix Algorithm*, J.C.S.S., Vol. 23, (1), 1981, pp. 11-21.

[JLR83] J. P. JOUANNAUD, P. LESCANNE and F. REINIG, *Recursive Decomposition Ordering*, in I.F.I.P. Working Conference on Formal Description of Programming Concepts II, D. BJORNER Ed., North-Holland, 1983.

[JOL82] J. P. JOUANNAUD and P. LESCANNE, *On Multiset Ordering*, Inf. Proc. Lett., Vol. 15, (2), 1982.

[LES82] P. LESCANNE, *Some Properties of Decomposition Path Ordering, a Simplification Ordering to Prove Termination of Rewriting Systems*, R.A.I.R.O., Informatique Théorique, Vol. 16, 1982, pp. 331-347.

[LES83] (*a*) P. LESCANNE, *How to Prove Termination? An Approach to the Implementation of a New Recursive Decomposition Ordering*, Technical Report, Centre de Recherche en Informatique de Nancy, Nancy, France, July 1983;
(*b*) P. LESCANNE, *Computer Experiments with the REVE Rewriting System Generator*, in Proceedings of the 10th POPL Symposium, 1983.

[KNB70] D. E. KNUTH and P. BENDIX, *Simple Word Problems in Universal Algebra*, in Computational Problems in Abstract Algebra, J. LEECH Ed., Pergamon Press, 1970, pp. 263-297.

[MUS79] D. R. MUSSER, *Abstract Data Types Specification in the AFFIRM System*, Proceedings of the I.E.E.E. Conference on Specifications of Reliable Software, Cambridge, Ma., 1979, pp. 45-57.

[ODN78] M. O'DONNELL, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science, Vol. 58, Springer-Verlag, 1978.

[PLT78] D. PLAISTED, *A Recursively Defined Ordering for Proving Termination of Term-Rewriting Systems*, Report R-78-943, Department of Computer Science, University of Illinois, Urbana, Ill., 1978.

[PTS81] G. E. PETERSON and M. E. STICKEL, *Complete Sets of Reductions for some Equational Theories*, J.A.C.M., Vol. 28, (2), 1981, pp. 233-264.

[ROS73] B. K. ROSEN, *Tree-Manipulating Systems and Church-Rosser Theorems*, J.A.C.M., Vol. 20, (1), 1973, pp. 160-187.