An Implementation of the KNS Ordering

J.M. Rivero

Report LSI–91–40

An Implementation of the KNS Ordering

J.M. Rivero

Universidad Politécnica de Cataluña
Dept. Lenguajes y Sistemas Informáticos
Pau Gargallo 5, E-08028 Barcelona, Spain
E-mail: rubio@lsi.upc.es

Draft. (April 1991)

Abstract: We describe an inplementation of the KNS ordering within the TRIP system [Nie 90, NOR 90], a Quintus-Prolog written laboratory for experimenting with new rewrite-like approaches to theorem proving in first-order logic. The aim of the TRIP system is not to obtain high-performance theorem provers, but rather to test techniques that could be the basis for such theorem provers.

The KNS ordering [KNS 85] is a partial ordering scheme for proving the uniform termination of term rewriting systems, which is a crucial aspect of Knuth-Bendix completion-like theorem proving procedures. The KNS ordering is used within TRIP as a subsystem of a very general completion procedure for first-order clauses with equality, based on recent work done in this field by Bachmair and Ganzinger (1990,1991) subsuming and generalizing many known results.

Our system contributes to the further development of these methods in several ways, and especially w.r.t. the definition of new techniques which are necessary for exploiting in their full power the redundancy notions that have now become available. We apply results from [NR 91], where a highly simplified framework for defining Knuth-Bendix-like completion procedures for full first order clauses with equality is defined and instantiated by a new, simpler and more restrictive inference system. This implementation uses the notion of *clausal rewriting* (Nieuwenhuis and Orejas, 1991) for the definition of a single formalized method for proving the redundancy of clauses and inferences.

1. Introduction

Knuth-Bendix-like completion [KB 70, Rus 87, HR 89, BG 90,91, NO 91] can be seen as a refutationally complete process that transforms a set of axioms in such a way that, by using the final *complete* set of axioms, efficient *normal form* proofs can be obtained (e.g. *linear* proofs, or the rewrite proofs in the equational case). The possibility to use powerful simplification techniques makes completion methods especially interesting for theorem proving and many other applications.

The successful use of the completion procedures crucially depends upon the ability to prove ordering relations between terms. The reason for this fact is that the inference rules applied, such as (strict) superposition or other forms of ordered paramodulation reduce the search space by selecting only the maximal literals and the maximal terms to paramodulate upon. More powerful mechanisms for eliminating candidates of maximal literals and terms therefore implies less inferences to be computed, which in turn means more efficiency in theorem proving, and more cases in which complete systems can be obtained. Moreover, more powerful orderings also allows more simplification and redundancy proofs to be possible, which again has the same advantages.

Our implementation of the KNS ordering has the ability to make *suggestions* for extending the underlying operator ordering in such a way that a certain pair of terms can be ordered. This has many advantages when trying to obtain complete systems, since this may involve some search, as sometimes complete systems exist for given operator precedences, but not for other ones. Moreover, it allows to extend the operator precedence incrementally, which has similar advantages.

In this note we will not describe the theoretical foundations of the KNS ordering. For details we refer to the original paper [KNS 85].

The organization of this note is the following. We first give some basic definitions, in section 2. In section 3 we give some examples of the application of our KNS subsystem, showing cases of incrementality, suggestions, and power. Finally, section 4 is devoted to a very brief description of the implementation.

2. Basic notions and terminology

First-order terms are denoted by s, t, t' etc. Substitutions are denoted by σ, σ', etc, and their application to a term t by $t\sigma$. Occurrences in terms are denoted by u, u', etc. If u is an occurrence of a subterm s in a term t, then t/u denotes s, and $t[u \leftarrow t']$ denotes the result of the replacement in t of the subterm at occurrence u by t'. The set of variables of a term t is denoted by $vars(t)$.

3. Examples

In this section we will show by means of two examples the kind of results that can be obtained by executing the algorithm. We first give some information to understand how the algorithm proceeds. Then we include the information displayed during the algorithm execution.

– The *precedence relationship* establishes several operator classes –one or more operators with equal precedence and equal status–, and "greater" precedence relationships between these classes. For each operator, `L` denotes the set of operators with lower precedence, and `E` the set of those with equal precedence.

– An *alternative* is a different operator precedence relationship and/or status assignment.

– The algorithm proposes a *status* associated to each operator symbol. It can be one of the following: `ud` (undefined), `ms` (multiset), `lr` and `rl` (lexicographic left to right and right to left respectively).

The user may propose an initial operator relationship which is respected and improved in order to obtain distinct ordering alternatives. This, for example, allows to work with a non-fully defined precedence during a completion procedure. The precedence will be extended as the procedure needs to specify more operator relationships.

Note that in order to obtain a total ordering between ground terms, equal precedence relationships between two operators are not allowed, unless the two operators are the same. Moreover, the precedence ordering between operators –function symbols or constants– should be total. In the first example give here we do not establish previous conditions on the operators, so we obtain the list of all posible alternatives.

An extension of this algorithm allows to work with general first order clauses with equality. Procesing them we can obtain all different alternatives for each division of the clause in maximal and nonmaximal literals. By this way, we may keep the number of maximal literals as low as posible and reduce the search space.

In the first example RPO does not apply if the precedence is $\mathtt{g} > \mathtt{h}$ (with `f` multiset status). However, within our KNS implementation, this is the only posibility suggested for a left-to-right ordering of these terms. The right-to-left ordering presents up to six alternatives depending on operator precedences and status.

In the second example we can see how the algorithm shows diferent alternatives to incrementally improve our initial precedence $\mathtt{f} > \mathtt{h}$, in order to orient that terms. We can see the high number of different suggestions the algorithm is able to provide. The reason is that, the more powerful the ordering is, the higher the number of alternatives presented, which allows to choose the most "reasonable" one to proceed.

Example 1:

```
?- orient_equation( f(g(A), g(B)) , f(h(A, B), h(A, B)) ).

Terms to be ordered :

      f(g(A), g(B))   vs.   f(h(A, B), h(A, B))

Previous operators relation :

 0.  f ud, ,                g ud, ,                h ud, ,

Left-to-Right alternatives :

 1.  f ms, ,                g ud, L= h,            h ud, ,

Right-to-Left alternatives :

 1.  f ud, ,                g ud, ,                h ud, L= g,
 2.  f ud, ,                g ms, , E= h           h ms, , E= g
 3.  f lr, L= g h,          g lr, , E= h           h lr, , E= g
 4.  f lr, , E= g h         g lr, , E= f h         h lr, , E= f g
 5.  f rl, L= g h,          g rl, , E= h           h rl, , E= g
 6.  f rl, , E= g h         g rl, , E= f h         h rl, , E= f g
```

Example 2:

```
?- orient_equation( f(A, h(A, B)) , g(B, f(B, A)) ).

Terms to be ordered :

      f(A, h(A, B))   vs.   g(B, f(B, A))

Previous operators relation :

 0.  f ud, L= h,              g ud, ,              h ud, ,

Left-to-Right alternatives :

 1.  f ms, L= g,            g ud, ,                h ud, ,
 2.  f rl, L= g,            g ud, ,                h ud, ,
 3.  f ms, ,                g ud, ,                h ud, L= g,
 4.  f rl, ,                g ud, ,                h ud, L= g,

Right-to-Left alternatives :

 1.  f ud, ,                g ud, L= f h,          h ud, ,
 2.  f ms, , E= g           g ms, L= h, E= f       h ud, ,
 3.  f rl, , E= g           g rl, L= h, E= f       h ud, ,
```

4. The implementation

In this section we show a small –and slightly simplified– part of the implementation of the algorithm. It only includes the part corresponding to the definition of the ordering. Other modules include the alternative reduction stages described below, the clause treatment mentioned above, user input-output, etc.

The algorithm works as follows. First an initial set of different alternatives is computed. These alternatives include all minimal extensions of the current precedence, such that the given pair of terms can be oriented in either direction. After this there are three stages of further reduction of this initial set of alternatives in order to keep only the "interesting" ones. This is necessary because otherwise the amount of information to be processed by the user would be too large.

The first and third stages eliminate alternatives that are subsumed by other ones. The second stage tries to compress groups of three alternatives into one simpler one. This can be done if the three alternatives differ only in the status assignment for a given operator `f`, and these assignments are `ms`, `lr` and `rl` respectively. The merged alternative obtained contains the common part of the given three alternatives, and leaves the status assignment for `f` undefined (`ud`).

```
/*              Basic predicates are greater_term and greater_path.

        greater_term(+Term1, +Term2)

        _ms is the multiset extension of a Predicate (term_ordering,
        full_paths, ...). Before calling an ordering_ms it's free-timing
        to delete repeated items of both multisets.
*/


greater_term(S, _T) :-
        var(S), !,                                  % S is una variable.
        fail.
greater_term(S, T) :-
        var(T),
        free_of_var(T, S), !,                       % T is a variable which
        fail.                                       % doesn't appair in S.
greater_term(_S, T) :-
        var(T), !.

greater_term(S, T) :-
        S =.. [F|_Si],
        T =.. [G|Ti],
        greater_operator(F, G),                     % The precedence of F
        delete_common_terms([S], Ti, Sd, Td),       % is greater than that of G.
        greater_term_ms(Sd, Td).

greater_term(S, T) :-
        S =.. [F|Si],
        T =.. [G|Ti],
        multiset_status(F),                         % F and G are multiset
        multiset_status(G),                         % status operators with
        equal_operator(F, G),                       % equal precedence.
        full_paths_ms(Si, P),
        full_paths_ms(Ti, Q),
        delete_common_paths(P, Q, Pd, Qd),
        greater_path_ms(Pd, Qd).
greater_term(S, T) :-
        S =.. [F|Si],
        T =.. [G|Ti],
        lexicographic_status(D, F),                 % F y G are lexicographic
        lexicographic_status(D, G),                 % status operators with
        equal_operator(F, G),                       % equal precedence.
        greater_term_lx(D, Si, Ti, S, T).

greater_term(S, T) :-
        S =.. [F|_Si],
        T =.. [G|_Ti],
        non_greater_non_equal_O_flag(F, G),         % The precedence of F isn't
        full_paths(S, P),                           % greater than that of G.
        full_paths(T, Q),
        greater_path_ms(P, Q).
```

```
/*
        greater_path(+Path1,  +LeftContext2, +RightContext2)
                     \- P -/  \------------ Q ------------/

        Only paths ending in the same variable can be compared.
        Before trying to compare two paths, they're dropped out
        their common suffixes, if any, in order to a more eficient
        comparison.

*/

greater_path([], _L2, _Q) :- !,
        fail.
greater_path(_P, _L2, []) :- !.                 % Trivial cases.

greater_path(P1, L2, [G~T|R2]) :-
        append(_L1, [F~_S|_R1], P1),
        greater_operator(F, G),                 % The precedence of F
        append(L2, [G~T], P2),                  % is greater than that of G.
        greater_path(P1, P2, R2).

greater_path(P1, L2, [G~T|R2]) :-
        append(L1, [F~S|R1], P1),
        multiset_status(F),                     % F and G are multiset
        multiset_status(G),                     % status operators with
        equal_operator(F, G),                   % equal precedence.
        greater_path_b1b2b3(L1, S, R1, L2, T, R2).

greater_path(P1, L2, [G~T|R2]) :-
        append(L1, [F~S|_R1], P1),
        lexicographic_status(D, F),             % F y G are lexicographic
        lexicographic_status(D, G),             % status operators with
        equal_operator(F, G),                   % equal precedence.
        greater_path_c1c2(L1, S, L2, T),
        append(L2, [G~T], P2),
        drop_common_suffix(P1, P2, R2, P1p, L2p, R2p),
        greater_path(P1p, L2p, R2p).


greater_path_b1b2b3(_L1, _S, R1, _L2, _T, R2) :-
        drop_common_suffix(R1, R2, R1p, R2p),   % RightContext of P greater
        greater_path(R1p, [], R2p).             % than RightContext of Q
greater_path_b1b2b3(_L1, S, R1, _L2, T, R2) :-
        drop_common_suffix(R1, R2, R1p, R2p),   % RightContext of P equal
        equal_path(R1p, R2p),                   % to RightContext of Q, and
        greater_term(S, T).                     % term S greater than T.
greater_path_b1b2b3(L1, S, R1, L2, T, R2) :-
        drop_common_suffix(R1, R2, R1p, R2p),   % RightContext of P equal
        equal_path(R1p, R2p),                   % to RightContext of Q, term
        equal_term(S, T),                       % S equal to T and
        drop_common_suffix(L1, L2, L1p, L2p),   % LeftContext of P greater
        greater_path(L1p, [], L2p).             % than LeftContext of Q.


greater_path_c1c2(_L1, S, _L2, T) :-
        greater_term(S, T).                     % Term S greater than T.
greater_path_c1c2(L1, S, L2, T) :-
        equal_term(S, T),                       % Term S equal to T and
        drop_common_suffix(L1, L2, L1p, L2p),   % LeftContext of P greater
        greater_path(L1p, [], L2p).             % than LeftContext of Q.
```

Acknowledgements: This work has been partially supported by a grant from the Esprit Project ICARUS.

5. References

[BDP 89] L. Bachmair, N. Dershowitz, D. Plaisted: Completion without failure. In H. Ait-Kaci and M. Nivat, editors, Resolution of equations in algebraic structures, vol 2: Rewriting Techniques, pp 1-30, Academic Press, (1989).

[BG 90] L. Bachmair, H. Ganzinger: On restrictions of ordered paramodulation with simplification. In Proc. 10th Int. Conf. on Automated Deduction. Kaiserslautern, 1990. LNCS, pp 427-441.

[BG 91] L. Bachmair, H. Ganzinger: Completion of first order clauses with equality. (final version) 2nd Intl. Workshop on Conditional and Typed Term Rewriting, Montreal (1991). To appear in LNCS.

[HR 89] J. Hsiang, M. Rusinowitch: Proving refutational completeness of theorem proving strategies: The transfinite semantic tree method. Submitted for publication (1989).

[KB 70] D.E. Knuth, P.B. Bendix: Simple word problems in universal algebras. J. Leech, editor, Computational Problems in Abstract Algebra, 263-297, Pergamon Press, Oxford, 1970.

[KNS 85] D. Kapur, P. Narendran, G. Sivakumar: A path ordering for proving termination of term rewriting systems. In Proc. Mathematical Foundations of Software Development, TAPSOFT 85, Berlin. LNCS 185.

[Nie 90] R. Nieuwenhuis: Theorem proving in first order logic with equality by clausal rewriting and completion. PhD thesis, UPC Barcelona, 1990.

[NO 91] R. Nieuwenhuis, F. Orejas: Clausal Rewriting. 2nd Intl. Workshop on Conditional and Typed Term Rewriting, Montreal (1991). To appear in LNCS.

[NOR 90] R. Nieuwenhuis, F. Orejas, A. Rubio: TRIP: an Implementation of Clausal Rewriting. In 10th Int Conf. on Automated Deduction, Kaiserslautern, 1990. Procs. in LNCS 449.

[Rus 87] M. Rusinowitch: Theorem-proving with resolution and superposition: an extension of Knuth and Bendix procedure as a complete set of inference rules. Report 87-R-128, CRIN, Nancy, 1987.