INSTITUTUL
DE
MATEMATICA

INSTITUTUL NATIONAL
PENTRU CREATIE
STIINTIFICA SI TEHNICA

ISSN 0250 3638

STRUCTURE SHARING WITH LITERAL INDEXING FILTER FOR RESOLUTION SYSTEMS

by

Dan NICOLĂIȚĂ

PREPRINT SERIES IN MATHEMATICS
No. 33/1987

BUCURESTI

STRUCTURE SHARING WITH LITERAL INDEXING FILTER FOR RESOLUTION SYSTEMS

by

Dan NICOLĂIȚĂ*)

September 1987

*) *Department of Robotics and Artificial Inteligence, Institute for Computers and Informatics, Bd. Miciurin nr. 8-10, 71316 Bucharest 1, Romania.*

1. INTRODUCTION

The effect of structure-sharing schemes on fundamental resolution theorem-proving algorithms was illustrated by prominant authors in domain [1], [12-14].

Beside the necessarily memory space economy achived using structure-sharing, there are other important aspects which essentially depend on whether or not the substitutions resulted during unification are applied on the respective terms/literals. Thus, the structure-sharing developed by Overbeek et. al.([13], [14]) is based on maintaining (in the permanent data structures) of only one copy for each generated term and making all containing terms, literals, clauses accesible from it. The unique feature of this scheme consists in its facility which allow simultaneously generating sets of (hyper-)resolvents ([12]) or paramodulants using a single unification, coupled with complete forward/backward subsumption checks (see also [10d,e]).

Early experience with resolution systems showed that an important part of the overall theorem-prover execution time was consumed by the searching process for finding unifiable and subsumable literals. Succesful attemps were made to eliminate most sequential searching by implementing retrieval mechanisms for rapidly locating of those literals that are potentially unifiable with or subsumed by a given literal. Significant exemples of such algorithms are reported in [3] (compatible formula outlines), [9], [11] (link inheritance and terminator module, a very fast procedure for finding unit refutations which can be also applied for literal indexing) and [13], [14] (unification properties, FPA lists).

This work presents a new structure-sharing scheme which is based on keeping a single copy for the rigid (functional, constant) part and flexible (variable) part of each object (term, literal), whereas the "containment" aspect remains valid. When an object is to be integrated into the database, its rigid and flexible parts must be searched (in paralell) to check if they are already integrated. The integration status of an object can be (in general) more rapidly decided if the search process is divided and oriented on the rigid and flexible parts of the respective object. Moreover, if the rigid part of the object was discovered as integrated then some important features about it can be directly obtained even if the object itself is not integrated, thus, there would be no need to re-build and re-attach them (these includes a variety of complexity measures,attributes and properties which characterize the rigid part). However, the main reason for employing this structure-sharing sheme is that each rigid part, actually designating a set of objects which includes alphabetic variants, can be viewed as a sort of global unification property. This, conceptually correspond to the method of Henschen-Naqvi ([3]), but the storage problems existing for the formula outlines are completely eliminated because (long-)integer identifiers are used to encode the integrated rigid parts. Calculating and linking the compatible rigid parts will facilitate the accessing of all potentially unifiable (forward, backward subsumable) candidates.

When performing an unification between two potentially unifiable literals l1 and l2, if the flexible part of l1 induces an unification conflict over the rigid part of l2 then the whole

set of literals {l2'} having the same rigid part as l2 has can be discarded from the unification test. The algorithm that check the existance of such an unification conflict is much less complex then the unification algorithm and it is sufficiently to call it a single time for the entire literal set {l2'}. The ability of discarding set of potentially unifiable candidates by unexpensive unification conflict checks represents another feature of the proposed structure-sharing scheme.

2. A FORMAL APPROACH

The methods contained in this chapter were inspired by the work of Henschen and Naqvi ([3]).

The definitions and the results are presented in the general frame of term algebra T ([5]), [16]), although they are still valid in the case of literal space.

The following notations are used:

- F, C, V the set of functional, constant and respectively variable symbols, M = F U C U V, N the set of natural numbers, $N^+ = N \setminus \{0\}$,
- P(W) the powerset of an arbitrary set W,
- a :T→ N the arity mapping,

$$a(t) = \begin{cases} n & \text{if } t = f(t1,\ldots,tn) \\ 0 & \text{otherwise} \end{cases}$$

- var : T→ P(V) the variable set mapping,

$$var(t) = \begin{cases} \emptyset & \text{if } t \in C \\ \{t\} & \text{if } t \in V \\ \bigcup_{j=1}^{n} var(tj) & \text{if } t = f(t1,\ldots,tn) \end{cases}$$

- d : T→ N the depth mapping,

$$d(t) = \begin{cases} 0 & \text{if } t \in C \cup V \\ 1 + \sum_{j=1}^{n} d(tj) & \text{if } t = f(t1,\ldots,tn) \end{cases}$$

- ds : T→ M the dominating symbol mapping,

$$ds(t) = \begin{cases} t & \text{if } t \in C \cup V \\ f & \text{if } t = f(t1,\ldots,tn) \end{cases}$$

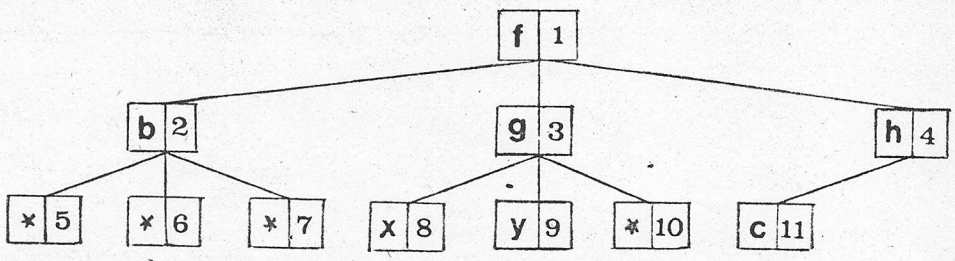
- pv : T→ $P(\bigcup_{j=1}^{\infty} N^j)$ the mapping which associate to each term the set of its position vectors; if $l \in pv(t)$ then t/l designate the subterm of t starting at the position vector l.
- pri generally denote the i-order projection;

$m = \max\{a(f) \mid f \in F\}$, we assume $m < \infty$.

2.1. Terms as m-ary complete trees

In this section the terms are represented through m-ary complete trees. Such a representation is, in general, expensive when m and/or the depth nesting of the terms has relatively large values ([3]). However, the actual need to consider terms as m-ary complete trees is only for motivating the indexing mechanism; in practice, they will be represented as usual.

Example: Let m = 3 and t = f(b, g(x,y), h(c)).



It can be observed that every occurrence of the term's symbols has its position associated into the nodes of the tree. Obviously, the nodes containing the * ($\notin$ M) symbol are irrelevant.

In subseqent it is necessary to deal with an application which transforms every term into a set of pairs consisting of term's symbols occurrences and their positions from the above representation. Therefor, let us define the folowing mappings:

- # : (M x N^+) x $N^+ \to$ M x N^+, (s,n) # n' = (s,n+n') $\forall$ (s,n)$\in$ M x N^+, n' $\in N^+$.
- L # n' = {(s,n+n')|(s,n) $\in$ L} $\forall$ L$\subset$ M x N^+, n'$\in N^+$.

$$- sp' : T \times N \to P(M \times N^+),\ sp'(t,d') = \begin{cases} \{(ds(t),1)\} & \text{if } d' = 0 \\ U_{i=1}^{n} (sp'(ti,d'-1)\#i\cdot m^{d'-1}) & \text{if } t = f(t1,\ldots,tn) \text{ and } d' \geq 1 \end{cases}$$

- $sp : T \to P(M \times N^+)$, $sp(t) = U_{d'=0}^{d(t)} sp'(t,d')$; this application transforms t into a set of pairs of type (symbol, position) and being injective it induces a non-ambiguously representation for terms.

Evidently, for each t $\in$ T there is a bijection p $\longmapsto$ lp between pr2(sp(t)) and pv(t).

2.2. Algebraic properties

Definition: Let t $\in$ T and RPt = {(s,p) $\in$ sp(t)|s$\in$ F U C}, FPt = sp(t)\ RPt. RPt and FPt are called the rigid part and respectively the flexible part of the term t.

We observe that RPx = ϕ $\forall$ x $\in$ V and FPt = ϕ $\forall$ t $\in$ T with var(t) = ϕ. Let t1,t2 $\in$ T and t1 ~rp t2 iff RPt1 = RPt2, t1 ~fp t2 iff FPt1 = FPt2. The relation "~rp" and "~fp" are congruence relations on T. Let K = {RP|RP = RP$\hat{t}$, $\hat{t} \in$ T/~rp}, K' = {FP|FP = FP$\bar{t}$, $\bar{t} \in$ T/~fp}, (RP$\hat{t}$ = RPt' $\forall$ t'$\in \hat{t}$, FP$\bar{t}$ = FPt'' $\forall$t''$\in \bar{t}$, $\forall$t $\in$ T), K and K' can be organized as complete algebras of the same type as T by putting f(RP1,...,RPn) = RP$\hat{t}$ where RPi = RP$\hat{t}$i $\forall$ $\forall$ i = $\overline{1,n}$, t = f(t1,...,tn), $\forall$ f$\in$ F (analogously for K'). K is izomorphic with T/~rp and K' is izomorphic with T/~fp.

For every $f \in F$, the f-section of T,K and K' is defined as $Tf = \{t \in T \mid ds(t) = f\}$, $Kf = \{RP \mid RP = \underline{RP\hat{t}},\ t \in Tf/\tilde{}rp\}$ and respectively $K'f = \{FP \mid FP = FP\bar{t},\ t \in Tf/\tilde{}fp\}$. Obviously, $(Tf)_{f \in F}$ and $(Kf)_{f \in F}$ are partitions of T and respective K.

Let $f \in F$ and $kf : Kf \longrightarrow K^{a(f)}$, $k'f : K'f \longrightarrow (K')^{a(f)}$ defined as: $kf(RP) = (RP1,\ldots,RPn)\ \forall\ RP \in Kf, RP = RP\hat{t}, t = f(t1,\ldots,tn)$, $RPi = RP\hat{ti}\ \forall\ i = \overline{1,n}$,
$k'f(FP) = (FP1,\ldots,FPn)\ \forall\ FP \in K'f, FP = FP\bar{t}, t = f(t1,\ldots,tn)$, $FPi = FP\bar{ti}\ \forall\ i = \overline{1,n}$. kf and k'f are injective mappings $\forall\ f \in F$ and this means that the rigid and flexible parts of any term are uniquely determined by the rigid and flexible parts of its arguments taken in the respective order.

2.3. The notion of integrated terms

Let $T^* \subset T$, $C \cup V \subset T^*$, $\forall\ f \in F\ \exists t \in T$ with $ds(t)=f$ and $t \in T^*$

Let $sid : M \longrightarrow N$ be an injective mapping; we assume that the followimg injective mappings were defined: $rpid : T^*/\tilde{}rp \longrightarrow N$, $fpid : T^*/\tilde{}fp \longrightarrow N$ having the properties $rpid(RP\hat{c}) = sid(c)\ \forall\ c \in C$, $rpid(RP\hat{x}) = 0\ \forall\ x \in V$, $fpid(FP\bar{x}) = sid(x)\ \forall\ x \in V$, $fpid(FP\bar{c}) = 0\ \forall\ c \in C$.
Evidently, rpid and fpid are inducing an injective application on T^* denoted by $tid : T^* \longrightarrow N^2$, $tid(t) = (rpid(RP\hat{t}), fpid(FP\bar{t}))\ \forall\ t \in T^*$.
For every $f \in F$, the following order relations are defined on $T^*f/\tilde{}rp$, $T^*f/\tilde{}fp$ and $T^*f\ (n = a(f))$:
- $RPt1 \leqslant RPt2$ iff $(rpid(pr1(kf(RP\hat{t1}))),\ldots,rpid(prn(kf(RP\hat{t1}))))$ $\leqslant_{lex}$ $(rpid(pr1(kf(RP\hat{t2}))),\ldots,rpid(prn(kf(RP\hat{t2}))))$, $\forall\ \hat{t1},\hat{t2} \in T^*f/\tilde{}rp$.
- $FPt1 \leqslant FPt2$ iff $(fpid(pr1(k'f(FP\bar{t1}))),\ldots,fpid(prn(k'f(FP\bar{t1}))))$ $\leqslant_{lex}$ $(fpid(pr1(k'f(FP\bar{t2}))),\ldots,fpid(prn(k'f(FP\bar{t2}))))$, $\forall\ \bar{t1},\bar{t2} \in T^*f/\tilde{}fp$.
- $t1 \leqslant t2$ iff $tid(t1) \leqslant_{lex} tid(t2)\ \forall\ t1,t2 \in T^*$.

(" $\leqslant_{lex}$" means the usual lexicographic order).

Consider now $(Q, \leqslant)$ an arbitrary ordered set and the boolean procedure search(q,Q) returning true if q is found in Q and false otherwise; if $q \notin Q$ then q is inserted into Q by the procedure Insert(q,Q).

Definition. Let $t \in T \setminus (C \cup V)$, t is called T^*-integrated if:
(1) the arguments of t are T^*-integrated,
(2) the procedures Search($RP\hat{t}$, $T^*ds(t)/\tilde{}rp$), Search($FP\bar{t}$, $T^*ds(t)/\tilde{}fp$) and Search(t, $T^*ds(t)$) all return true.

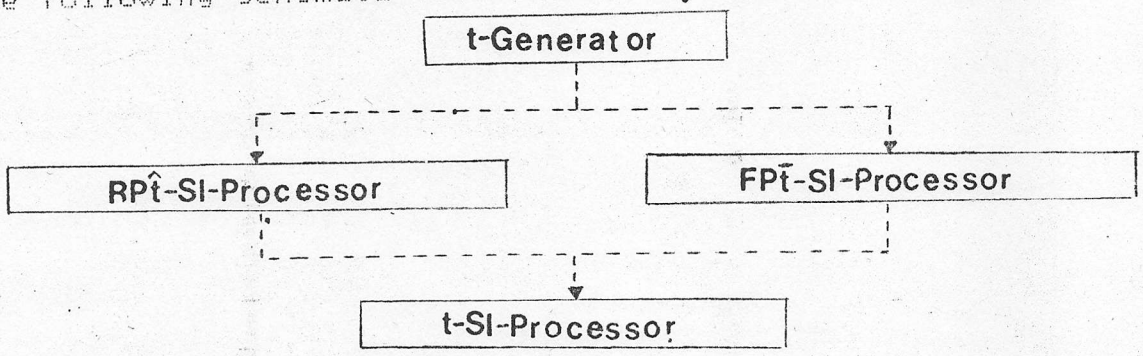
A theorem-prover can be viewed as a generator of terms (literals) which will be maintained in the permanent data structure. It is very important (from the memory space perspective) to keep a single copy for each term (literal) that is generated. If we formally agree to maintain the generated terms into a set T^* then the problem reduces to keep T^* as a set and not as a multiset. Thus, before integrating (including) a composite term t into T^* it is necessary to check if t is not already in T^*. This check will be firstly applied to $RP\hat{t}$ and $FP\bar{t}$ and then to t.

The integration procedure Integrate(t,T^*):
(1) let $t = f(t1,\ldots,tn)$ and args := $\{t1,\ldots,tn\}$
(2) while (args $\neq \phi$) do
begin let t' := first(args); if $t' \notin C \cup V$ then

```
                Integrate(t', T*);
                args := args \ {t'}
        end

(3) if Search(RPt̂, T*f/~rp) returns true
        then GOTO (4)
        else LastRPId := LastRPId+1; rpid(RPt̂) := LastRPId;
             Insert(RPt̂, T*f/~rp)
(4) if Search(FPt̄, T*f / ~fp) returns true
        then GOTO (5)
        else LastFPId := LastFPId+1; fpid(FPt̄) := LastFPId;
             Insert(FPt̄, T*f/~fp)
(5) if Search(t,T*f) returns false then Insert(t, T*f).
    (Initially, LastRPId := max{sid(c)|c ∈ C},
                LastFPId := max{sid(x)|x ∈ V})
```

The integration procedure employs a structure-sharing scheme by maintaining a single copy for each term and its rigid-flexible parts. The operations of this procedure can be summarized by the following schemata ("SI" abbreviates Searching- Insertion):



As could be observed, unique integer identifiers are not assigned to terms (as in [13], [14]) but instead to their rigid and flexible parts.

2.4. Potential unifiability and subsumability

Definition. Let t1,t2 ∈ T and RPt1, RPt2 the rigid parts of these terms.
(1) t1 is p(potential)-unifiable with t2 if ∀ (s1,p) ∈ RPt1 and ∀ (s2,p) ∈ RPt2 implies s1 = s2; RPt1 and RPt2 are called u-compatible.
(2) t1 p(potential)-subsumes t2 if RPt1 ⊆ RPt2; RPt1 is said s-compatible with RPt2.

Evidently, the u-compatible relation is reflexive and simetric while the s-compatible relation is an order relation on K.

Theorem 1. For every t1,t2 ∈ T, the following implications hold:
(1) t1 is unifiable with t2 ⟹ t1 is p-unifiable with t2 i.e. RPt1 is u-compatible with RPt2.
(2) t1 subsumes t2 ⟹ t1 p-subsumes t2 i.e. RPt1 is s-compatible with RPt2.

A similar result is obtained in [3] and it represents the filter for the retention of those candidates which potentially unify with (subsume) a given term or literal.

Theorem 2. Let the applications ku, ks : $K \longrightarrow P(K)$ be defined as:
$ku(RP) = \{RP' \mid RP' \text{ is u-compatible with } RP\} \ \forall RP \in K$,
$ks(RP) = \{RP' \mid RP' \text{ is s-compatible with } RP\} \ \forall RP \in K$.
Then, ku and ks are injective.

This theorem shows that the rigid part $RP\hat{t}$ generated by an arbitrary (composite) term $t \in T$ is uniquely determined by the set of all rigid parts u(s)-compatible with $RP\hat{t}$.

Theorem 3. Let $RP \in K$, $RP = RP\hat{t}$, $t \in T \setminus (C \cup V)$, $ds(t) = f$, $a(f) = n$. Then:
(1) $ku(RP) = \bigcap_{i=1}^{n} \{RP' \in Kf \mid pri(kf(RP')) \text{ is u-compatible with } pri(kf(RP))\}$
(2) $ks(RP) = \bigcap_{i=1}^{n} \{RP' \in Kf \mid pri(kf(RP')) \text{ is s-compatible with } pri(kf(RP))\}$.

This theorem indicates the method of building the set of all rigid parts u(s)-compatible with a given rigid part, starting from the rigid parts of the arguments. Because each intersected set can be ordered, using as an ordering key the rpid of the RPs which it contains, the intersection operation itself is performed as for usual ordered sets. This resembles with [13] and it is different from [3] which uses standard string operations to check u(s)-compatibility.

Theorem 4. Let t1,t2 $\in$ T, t1 is assumed p-unifiable with t2, and let $E = pr2(FPt1) \cap pr2(RP\hat{t}2) \neq \emptyset$. Suppose $\exists$ x $\in$ var(t1) and $\exists p1,p2 \in E$ with $(x,p1),(x,p2) \in FPt1$.
If $t1' = t2 / lp1$, $t2'' = t2 / lp2$, $t1', t2'' \in (T \setminus V)$ and t1' is not p-unifiable with t2'' then for every $t \in \hat{t2}$, t is not unifiable with t1 (in particular t2 is not unifiable with t1).

This theorem gives a sufficient condition under which a whole set of potential candidates ($\hat{t2}$) can be discarded from the unification test with a given term (t1).

3. Implementation

Each term or literal (object) is uniquely characterized by its rigid and flexible parts.

A structure-sharing scheme can be developed by maintaining only one copy of any object and making all containing terms, literals, clauses or lists accesible from it ([13],[14],[10d,e]). Although the containment aspect remains valid, there is another alternative which consists in keeping single copies for the rigid and flexible parts of all objects in the knowledge database. When we are interested to integrate (in the database) an object, its rigid and flexible parts must be searched (eventually in parallel) to check if they are already integrated. For this purpose, two (long) integer identifiers (rpid, fpid) are assigned to each object, corresponding to its rigid and flexible parts. Because these two identifiers are both starting from 0, we have to consider them as long integers (fixed length arrays of integers) only for applications with very large number of generated objects. I believe that the integration status of an object can be more rapidly decided if the search process is divided and oriented on the rigid and flexible parts of the respective object. It is sufficient to discover that the rigid or

flexible part is not integrated, to conclude that the object is surely not integrated. On the other hand, if the rigid part of the object is integrated we can automatically obtain some important information about it even if the object itself is not integrated. These includes a variety of complexity measures([15]), attributes and properties ([14]) which are attached to sets of objects characterized by identical or compatible rigid parts. I have also some reasons to believe that introducing special compatibility definitions for rigid and flexible parts, algorithms that support intelligent backtracking can be implemented for this structure-sharing.

The unification properties of LMA system ([14]) which are assigned to each object, can be viewed as a description, in terms of position (integer) vectors, of its rigid part. If we take the conjunction of these unification properties, what we obtain is exactly the rigid part, which can be thus considered as a global unification property. Of course, there are other unification properties which must be taken into consideration. They result by replacing each occurence of the rigid (functional, constant) symbols of the respective object by a variable. It is known that the boolean combination of the unification properties induces a series of unions and intersections of the corresponding accessed FPA lists which return unification candidates (by the "get-first", "get-next" functions). The same result is obtained if we take a series of unions and intersections of lists whose nodes are u-compatible rigid parts. However, the number of these operations and the length of the lists involved is smaller because of the using of global unification properties. Generating all unification/subsumption candidates may create some overhead, but once this happens, we don't have to retrieve with efforts the needed candidates for any object whose rigid part is integrated, no matter how often it is used in the general unification process. Since objects are grouped under the same rigid parts and the u-compatible rigid parts are linked, the candidates are accessed in this form (which is similar to [3]). As a consequence, there is another aspect which might not be out of interest:

There are cases when we don't need just an arbitrary unification candidate (for a given literal) but one that has a designated attribute (lock value ([2],[8]), key, sorted or qualifier literal ([17],[19],[20]), significant subterms ([4],[21]), complexity measure ([15])). Such a candidate can be obtained by scanning the respective list of candidates until one with the desired attribute is found.

3.1. Basic data structures

In subsequence, an option for the data structures supporting the above structure-sharing is presented. These data sructures are enhanced to implement the mechanism for retrieving unification/subsumption candidates. Similar devices can be designed for paramodulation ([2],[4],[8],[21],[10b,c]) and dynamic demodulation ([7],[9],[15],[22],[10e]).

The following record type is used to represent terms and literals:

```
type TerLit   =  record
                    case tltype:(1,2,3) of
                    1:vid:integer;
                      clindex:SubstAr);
```

```
                    2:(cname:NameAr;
                       cid:integer);
                    3:(tl:(t,pl,nl,);
                       ds:^ DomSymb;
                         args:^ TerLitList;
                         contcl:^ ClList;
                       fpid:integer;
                         rp:^ RP)
                end.
```

The meaning of the fields in TerLit is:
tltype:this tag field indicates the type of the respective object:variable(1), constant(2), composite term or literal(3);
1:vid:the identifier of the variable;
clindex: a record structure used to index the variable (for simulating renaming) and to keep the bindings (substituents) resulted from the unification process);
2:(cname: the name of the constant;
cid: the identifier of the constant);
3:(tl: the value of this field is t if the object is a composite term, pl if it is a positive literal and nl if it is a negative literal;
ds: the pointer to a record structure containing informations about the object's dominating (major) symbol and the rigid-flexible parts of all objects admitting the respective dominating symbol;
args:the header to the list of arguments;
contcl: the header to the list of all containing clauses; this field is assigned to nil in the case of composite terms;
fpid: the identifier of the flexible part of respective object;
rp: the pointer to the rigid part (RP) of the object).

```
type SubstAr = array[1..max-clindex] of Subst;
type Subst   =   record
                   subst:^TerLit;
                   sindex:integer;
                   integr:boolean
                 end.
```

These two record structures are used by the unification algorithm (see 3.4). It is considered that the maximum number of parent clauses for a (hyper-)resolvent is max-clindex. To overcome this restriction we can implement SubstAr as a linked list of fixed size arrays. To each parent clause we assign an (integer) index (from 1 to max-clindex) which is used to enter into SubstAr and to find the substituent (given by subst field) of the respective variable. The sindex field indicates the index of the substituent and the integr field (used to efficientizize integration) indicates if the substituent is integrated or not (see [10d]).

```
type DomSymb    =  record
                     sname:NameAr;
                     sid,arity:integer;
                     prfp,nrfp:^RFPInfo
                   end.

type TerLitList = record
                     terlit:^TerLit;
```

```
                    next:^TerLitList
                  end.
```

The meaning of the fields in DomSymb is:

sname: the name of the respective dominating (functional, predicative) symbol;

sid: the identifier of the dominating symbol;

arity: the arity of the dominating symbol;

prfp,nrfp: the pointers to two record structures containing informations about the rigid-flexible parts of all objects admitting the respective dominating symbol; in the case of functional symbols, nrfp is assigned to nil; in the case of predicative symbols, prfp correspond to all positive literals and nrfp to all negative literals of respective predicate.

```
type RFPInfo = record
                 rptree:RPTreeAr;
                 fptree:FPTreeAr;
                 integrtree:IntegrTreeAr;
                 const:^ArgRPList1;
                 varg:^ArgRPList2;
                 vrp:^RP
               end.
```

The meaning of the fields in RFPInfo is:

rptree: a hash table containing in each location the root of a binary, lexicographically ordered tree; each node of these trees contains the pointer to a rigid part (RP) corresponding to a set of objects, so that the nodes of all trees in this hash table contain all the rigid parts corresponding to the objects admitting the same dominating symbol; the key of the lexicographic order is given by the second field of type RPi.e. by the list of the rigid part identifiers associated to the arguments of the respective set of objects;

fptree: a hash table containing in each location the root of a binary, lexicographically ordered tree; each node of these trees contains the flexible part identifier and the list of the flexible part identifiers corresponding to the arguments of a set of objects; the rigid part (RP) of this set of objects is contained in a node of a tree from rptree; the previous list of fpids is the ordering key;

integrtree: a hash table containing in each location the root of a binary, lexicographically ordered tree; each node of these trees contains the pointer to an object (term/literal); the ordering key is obtained from the rigid and flexible part identifiers of the rspective object;

const: the root of an ordered list; the ordering key is given by the identifiers of the constant symbols; each node of this list contains (beside the ordering key) a header to another list; each node of this last list contains the header of the (containment) list of all those rigid parts (RPs) corresponding to the sets of objects that admit the respective constant as a designated argument;

varg: the header to the list which contain in each node the (containment) list of all those rigid parts (RPs) corresponding to the sets of objects that admit a variable as a designated argument;

vrp:the pointer to the rigid part (RP) corresponding to the set of objects that have only variables as arguments;

I emphasize that all objects referred above have the same dominating symbol,i.e., the dominating symbol described by the DomSymb record structure which was used to access ^RFPInfo through prfp, nrfp,fields.

```
type RPTreeAr = array[1..1] of ^RPTree;
type FPTreeAr = array[1..1] of ^FPTree;
type IntegrTreeAr = array[1..1] of^IntegrTree;
type RPTree = record                type FPTree = record
                                                    fpid:integer;
                rp:^RP;                             fpids:^IdList;
                left,right:^RPTree                  left,right:^FPTree
              end;                                  end;

type IdList = record                type IntegrTree = record
                id:integer                            terlit:^TerLit;
                next:^IdList                          left,right:^IntegrTree
              end.                                    end.
```

We can replace fpids in FPTree by (terlit:^TerLit) to save some memory space. In this case, to obtain the list of the arguments' flexible part identifiers we must access every fpid from the arguments of the object pointed by terlit.

The following record type encode the rigid part associated to a whole set of composite terms or literals:

```
type RP = record
            rpid:integer;
            rpids:^IdList;
            terslits:^TerLitList;
            pruc,fwsc,bwsc:^RPList;
            cont:^ArgRPList1
          end.
```

The meaning of the fields:

rpid:the identifier of the respective rigid part (RP);

rpids:the list of the rigid part identifiers corresponding to the arguments of the objects admitting this rigid part;

terslits:the header to the list of all objects that admit the respective rigid part (RP);

pruc:the header to the ordered list of all rigid parts (RP′) that are proper u-compatible with RP,i.e.RP′is u-compatible with RP but $RP' \not\subset RP$, $RP \not\subset RP'$ (the inclusion are only formally considered);

fwsc:the header to the ordered list of all rigid parts (RP′′) that are forward s-compatible with RP i.e.$RP'' \subset RP$;

bwsc:the header to the ordered list of all rigid parts (RP′′′) that are backward s-compatible with RP i.e.$RP \subset RP'''$;

cont:the root of an ordered list which facilitate access to containment lists; the ordering key is given by the identifiers of functional/predicative symbols; each node of this list contains the header to another list; each node of this last list contains the header to the list of all those super-rigid parts that admit RP as a designated argument and have the dominating symbol determined by the respective ordering key;

```
type ArgRPList1=record          type ArgRPList2=record
                   sid:integer;                    pos:integer;
                   arp:^ArgRPList2;                rpl:^RPList;
                   next:^ArgRPList1                next:^ArgRPList2
                 end;                            end;

        type RPList=record
                      rp:^RP;
                      next:^RPList
                      end.
```

The meaning of the fields of ArgRPList1:

sid: the identifier of a constant (when ^ArgRPList1 is used in type RFPInfo) or a functional/predicative (when ^ArgRPList1 is used in type RP) symbol; (we shall always consider that if id(P) is the identifier of a predicative symbol, then id(P)-1 is the identifier of ~P i.e. the identifier of the predicative symbol of a negative literal starting with P); the ArgRPList1 list is ordered using sid as an ordering key;

arp: the header to the list of all rigid parts corresponding to the sets of objects admitting as an argument the respective constant (given by the sid field, when ^ArgRPList1 is used in type RFPInfo) or rigid part (when ^ArgRPList1 is used in type RP); in the second case the sid field gives the dominating symbol of these sets of objects.

The meaning of the fields of ArgRPList2:

pos: the position of the respective argument (constant or rigid part) in the list of the arguments of the containment super-rigid parts from rpl;

rpl: the header to the containment list of rigid parts; this list is ordered on the rpid of each RP.

3.2. The integration algorithm

In sequel, the integration of objects in the knowledge database is briefly discussed (the technique is similar to [13],[10d,e]).

Let t=f(t1...tn) be an arbitrary term and LastRPId, LastFPId two global integer type variables. Because the integration algorithm is recursive we shall consider that t1,...tn are already integrated. Two hash keys (k1,k2) are obtained from the rigid part identifiers (rpids) and respectively the flexible part identifiers (fpids) of t1,...tn. Obviously, the rpid of a variable is 0 and the fpid of a constant or a ground term is 0. The rigid part and the flexible part of t is searched into the trees t^.ds^.prfp^.rptree[k1] and respectively t^.ds^.prfp^.fptree[k2]. The keys used in tree-searching are (t1^.rp^.rpid,...,tn^.rp^.rpid) and (t1^.fpid,...,tn^.fpid). If tj is a constant then its rpid is tj^.cid and its fpid is 0; if tj is a variable then its rpid is 0 and its fpid is tj^.vid. If the first (second) searching process fails then we can say that the RP(FP) of t is not integrated. If t is deemed appropiate to be added into the database then to obtain its rpid (fpid) we simply increment LastRPId (LastFPId). If both RP and FP of t were integrated, this does not necessarily mean that t is integrated. So, t has to be searched into t^.ds^.prfp^.integrtree[k] where k is a hash key obtained from the rpid and fpid of t. Of course, the searching key is (rpid,fpid). In case ^t,^RPt or ^FPt is found then it will be subsequently used, avoiding thus the

creation of multiple copies.

3.3 Generating u-compatible rigid parts

The following procedure returns all u-compatible RPs related to the RP of a given term ter. It is assumed that ter might have substituted variables, so that its RP will actually result by formally applying the respective substitution. This procedure can be used when performing hyperresolution [2], [8], [10a,c]. (For the implementation of this inference rule see [12], [14a] [10d]). The way it works and the meaning of the data structures it uses are presented below.

```
type IdAr = array[1..max-arity] of integer;
type CandArgAr = array[1..max-arity] of CandArg;
type CandArg = record
                    candtype:(v,c,r,rs);
                    ds:^DomSymb;
                    cid,pos:integer;
                    rp:^RP;
                    rps:^RPList
               end;

procedure UnifRPCands(ter: ^TerLit;i:integer;var rip:^RP;
rips:^RPList;find,stest:bolean);
label:1;
var t,tt:^TerLit;targs:^TerLitList; find1,stest1:bolean;
poss,nva:integer;idar:IdAr;candar:CandArgAr;s:Subst
begin find:=true; stest:=true;
 if ter^.fpid = 0
  then rip:=ter^.rp {ter is a ground term}
  else begin targs:=ter^.args;poss:=0;nva:=0;
   while targs< >nil do
    begin t:=targs^.terlit;poss:=poss+1;
     case t^.tltype of
      1: begin ReturnBinding(t,i,s);tt:=s.subst;
       if tt< >nil
        then case tt^.tltype of
          1:begin if find then idar[poss]:=0 end;
          2:begin if stest then begin stest:=false end;
                  if find then begin idar[poss]:=tt^.cid end;
                  nva:=nva+1; with candar[nva] do
                                  begin candtype:=c;
                                        pos:=poss;
                                        cid:=tt^.cid
                                  end
            end;
          3:begin UnifRPCands(tt,s.sindex,rip,rips,find1,stest1)
              if stest then begin stest:=false end;
              nva:=nva+1;
              if find1
               then begin with candar[nva] do
                             begin candtype:=r;
                                   rp:=rip;
                                   ds:=tt^.ds;
                                   pos:=poss
                             end;
                    if find
                      then idar[poss]:=rip^.rpid
                end
```

```
                else with candar[nva] do
                     begin candtype:=rs;rps:=rips;
                           ds:=tt^.ds;pos:=poss
                     end;
                  if find then find:=false
                end
               end{case}
            else if find then idar[poss]:=0
          end;
        2:begin nva:=nva+1; if find then begin idar[poss]:=t^.cid end;
                with candar[nva] do
                  begin candtype:=c; pos:=poss; cid:=t^.cid end
          end;
        3:begin UnifRPCands(t,i,rip,rips,find1,stest1);
                if stest then begin stest:=stest1 end; nva:=nva+1;
                if find1 then begin with candar[nva] do
                                  begin candtype:=r; rp:=rip;
                                        ds:=t^.ds; pos:=poss
                                  end; if find then idar[poss]:=rip^.rpid
                              end
                         else with candar[nva] do
                                  begin candtype:=rs;rps:=rips;
                                        ds:=t^.ds;pos:=poss
                                  end; if find then find:=false
          end
       end{case};targs:=targs^.next
     end{while}; if stest then rip:=ter^.rp
                          else begin if find
                                     then begin if nva=0
                                                then rip:=ter^.ds^.prfp^.vrp
                                                else begin RPTreeSearch
                                                (ter^.ds^.prfp^.rptree,idar,
                                                ,poss,rip);
                                                if rip=nil
                                                   then find:=false;GOTO1
                                                end
                                          end
                                     else GOTO 1
                                     end;
  end;
     1:begin Intersect(ter,nva,candar,rips) end
end.{UnifRPCands}
```

The recursive procedure UnifRPCands takes a term ter (of variable discrimination index i) and returns the pointer rip to its rigid part RP if RP is integrated, i.e. find = true. (Here, RP results after formally applying the substitution). In case RP is not integrated (find = false) this procedure returns the header rips to the corresponding RPList of all u-compatible rigid parts. Each argument of ter is visited and if find is true then the respective entry of idar is assigned to the argument's rigid part identifier.

For a variable argument, procedure ReturnBinding will facilitate accessing the Subst structure of it and therefore the substituent. The stest local variable remains true if no substituton was made on ter. If the rigid parts of all arguments are integrated then find will be true and there is a chance that the RP of ter itself is integrated. To check this, procedure RPTreeSearch(rptree : RPTreeAr; idar:IdAr; poss : integer; var rip:^RP) is used to search into the respective entry of RPTreeAr; idar contains the rigid part identifiers of all

arguments and poss is the number of them (actually, idar has a sufficiently large chosen size "max-arity" representing the maximum permitted arity for functional/predicative symbols). First, a hash key is obtained from idar which helps locating the entry into RPTreeAr, then idar is used as a lexicographically ordering key. If the search is successful then rip will be assigned to the ^RP contained in the respective found tree-node (otherwise it will be set to nil). The entries of idar are ceased to be assigned at the moment find becomes false (this correspond to the case when a (sub-)argument of ter was discovered as non-integrated). Because the RP of ter may not be integrated, precautions are needed to be taken; they materialize through the use of candar. For each non-variable argument tk of ter (nva represents their number), the candtype field will be assigned to:
- c if the argument is a constant,
- r if the argument's rigid part is integrated and to rs otherwise.

Also, candar[k].pos represents the position (place) of tk in ter^.args list, ds is assigned with the pointer to the DomSymb structure of tk whereas rp represents the pointer to tk's rigid part (if candar[k].candtype = r) and rps represents the list of all u-compatible RPs related to the rigid part of tk (if candar[k].candtype = rs).

When compatible forward subsumption RPs are needed, every variable argument will be taken into consideration and the candtype field of candar will be set to v (for these arguments).

If find remains false, then candar is used by the following Intersect procedure to colect the respective containment RPLists and to intersect them, thus returning the RPList of all u-compatible RPs corresponding to the RP of ter.

In sequel the next two procedures are used:

```
procedure SearchRP1(l1:^ArgRPList1; id,p:integer; var ll1:^RPList);
     begin if there is a node N of list l1 with N^.sid = id and
           there is a node M of list N^.arp with M^.pos = p
           then ll1:=M^.rpl
           else ll1:=nil
     end;

procedure SearchRP2 (l2:^ArgRPList2; p:integer; var ll2:^RPList);
     begin if there is a node M of list l2 with M^.pos=p
           then ll2:=M^.rpl
           else ll2:=nil
     end;
```

By {RPList} we'll mean the set of all those RPs forming RPList; when linking RPLists, the RPLList is used:

```
          type RPLList=record
                          rpl:^RPList;
                          next:^RPLList
                       end;
     type InterAr = array[1..max-arity] of ^RPLList;

procedure Intersect(ter:^TerLit;nva:integer; candar:CandArgAr;
var rips:^RPList);
var p,i,id:integer;prfp:^RFPInfo;rip:RP;interar:InterAr;
(1) i:=1; id:=ter^.ds^.sid; prfp:=ter^.ds^.prfp;
(2) while i ≤ nva do
     begin p:=candar[i].pos
          (21) if candar[i].candtype=c
               then SearchRP1(prfp^.const,candar[i].cid,p,l1);
```

```
                SearchPR2(prfp^.varg,p,l2); link l1 with
                l2 and assign the header of the resulting
                ^RPLList to interar[i];
        (22)    if candar[i].candtype=r
                then let rip:=candar[i].rp;
                SearchRP1(rip^.cont,id,p,l1);
                    if l1< >nil
                        then suppose l1={rp1,...,rpm}, for each
                          rpj,1 ⩽j ⩽m, let lj1:=rpj^.pruc,
                          lj2:=rpj^.fwsc,lj3:=rpj^.bwsc;
                          link l1,lj1,lj2,lj3, 1 ⩽ j ⩽ m, and
                          assign the header of the resulting
                          ^RPLList to interar[i]
                        else suppose {rip^.pruc} U {rip^.fwsc} U
                          {rip^.bwsc} = {rp1,...rpn}, for each
                          rpk, 1 ⩽ k ⩽n,
                          SearchRP1(rpk^.cont,id,p,lk1);
                          SearchRP2(prfp^.varg,p,l2);
                          SearchRP1(candar[i].ds^.prfp^.vrp^.cont,
                          id,p,l3);
                          link lk1,l2,l3, 1 ⩽ k ⩽ n and assign the
                          header of the resulting ^RPLList to
                          interar[i];
        (23)    if candar[i].candtype=rs
                      then suppose candar[i].rps={rp1,...,rpm}, for
                      each rpj, 1 ⩽ j ⩽ m.
                      SearchRP1(rpj^.cont,id,p,lj1);
                      SarchRP1(candar[i].ds^.prfp^.vrp^.cont,
                      ,id,p,l3);
                      SearchRP2(prfp^.varg,p,l2); link lj1,l2,l3,
                      1 ⩽ j ⩽ m, and assign the header of the
                      resulting ^RPLList to interar[i];
            i:=i+1
    end;
```

(3) only the first nva entries of interar are occupied; each of these entries contains a list of RPLists which is to be interpreted as a union of RP sets; by performing the intersection of these unions of RP sets, the rips list is build consisting of all those RPs that are u-compatible with the RP of the term ter (whose variables may be substituted).

(4) dispose the nodes of every RPLList from interar's entries

Procedure Intersect visits each entry of candar and depending on the found value of the candtype field (of the respective entry), it colects the pertinent containment lists which are linked and assigned to the same order entry of interar. Thus, the sections (21),(22),(23) of Intersect perform the following operations (we shall refer to an argument of a RP by actually meaning the argument of the corresponding set of terms):

(21) access the list l1 of all those RPs admitting the respective constant as a p-place argument; access the list l2 of all those RPs that have an arbitrary variable as a p-place argument; of course, the dominating symbol of RPs are the same as for ter.

(22) consider rip and try to access (using its cont field) the list l1 of all RPs that admit rip as a p-place argument and have the same dominanting symbol as ter has; if we are "fortunate" to find a non-nil l1 then l1,lj1,lj2,lj3, 1 ⩽ j ⩽m, will contain all RPs that have a p-place argument which is u-compatible with rip (note that in this case it is superfluous to

access 12,13 lists of the "else" part from the "if" statement); if l1 = nil then visit each node of the "pruc", fwsc" and "bwsc" lists of rip and access the respective containment lists; then access 12 (as in (21)) and 13 which is the list of all RPs that have a "vrp" p-place argument (vrp is a RP admitting only variables as arguments, vrp has the same dominating symbol as rip (=candar[i].ds); the rpid of vrp is exactly the identifier of its functional symbol; vrp is not contained in rip^.fwsc because otherwise it would has to be added into the fwsc field of each RP admitting the same dominating symbol).

(23) candar[i].rps = {rp1,...,rpm} represents all RPs that are u-compatible with the RP of the p-place argument of ter (resulted by formally applying the respective substitution); for each rpj, $1 \leqslant j \leqslant m$, access the containment list (as usual); access the containment list for "vrp" and for arbitrary variable p-place argument (through varg field).

If we were interested to retrieve the list of all those RPs that forward subsumed the RP of ter then for each p-place argument of ter which is a variable, interar[p] is assigned to a 12 type list; everything else remains same, exept in (22) only fwsc field is accessed (not pruc or bwsc).

It is clear now that the speed of the generating of u-compatible RPs will greatly depend on how efficient the intersection operation is performed. Because each RP contains in its terslits field a whole set of terms (literals) including the respective alphabetic variants, the length of the RPLLists in each entry of interar are hoped to have, in general, a reasonable size. Moreover, many of the lists accessed with SearchRP1, SearchRP2 procedures could result to be nil. However, applying the distributivity property of intersection over union will imply an explosion of operations. Take for instance, a most plausible example of intersecting 4 RPLLists each having 8 nodes, 4096 intersections would have to be performed. A solution for this problem is to simulate the merge of the lists linked in RPLLists. Thus, each entry of interar will not actually contain a RPLList, but a selection tree ([6]) whose leaves are the previously linked lists. The intersection can now be performed as for usual ordered lists, although $\log_2 n$ comparisons are needed to select the least element among n ordered lists.

The entry of interar giving the rpid of RP which is searched into the other entries, can dynamically change if we always choose to start searching with the rpid furnished by the selection tree having the smallest (updated) number of nodes at its leaves'lists. (For this, a new field is added into type ArgRPList2, indicating the number of nodes in rpl field).

It is far more convenient, to take interar as a global variable for procedure Intersect and to maintain a sufficiently large "seltrees" array which is used to represent the selection trees. (For extrem cases, this array can be implemented as a linked lists of fixed-size arrays; normally the selection trees will fit into one such array). Each entry of interar contains three location numbers into seltrees indicating the entries occupied by the root, first and last leaf of the respective tree. The nodes of each selection tree will actually appear in their inverse order, from the last leaf to the root.

As was seen, the previous algorithm returns all u-compatible rigid parts (for a given RP) and this is quite opposite to the "get-first", "get-next" LMA philosophy ([14a]). Although, the method presented herein necessitate the creation of all u(s)-compatible RPs for an integrated RP, there is no such need when

implementing hyperresolution. In this case, the "get-first", "get-next" functions could be implemented to return, as many as needed, positive literal type RPs which are u-compatible with the nucleus' negative descendent literal type RPs resulted after applying the substitutions. (Similarly, in the case of s-compatible RPs when checking to see if a generated hyperresolvent is forward subsumed). Thus, the above algorithm can be modified to use the "updatable pointers" technique of [14a] for maintaining "positions" between succesive calls of the "get-next" function. These positions will ultimately induce an order relation among a global but not usually totally created sequence of intersection operations. Although, only a subset of intersections, corresponding to a subset of positions, is effectively considered their number can also be large. In my oppinion, we can obtain a good overall speed for retrieving unification candidates if we resume to maintain positions only for the literal's arguments level. To be more explicit, in the case of u-compatible literal rigid parts, the algorithm will obtain all u-compatible RPs for the arguments (of the respective literal) and will maintain positions only when performing intersections between literal type RPLists.

When integrating the RP of a term (literal) t = f(t1,...,tn) we have to retrieve all u(s)-compatible RPs related to RP which are organized into three categories: "pruc", "fwsc" and "bwsc". The corresponding algorithm will be an Intersection type procedure where only (21) and (22) sections are considered. For each p-place argument of t which is a variable, interar[p] will contain the RPList of all those RPs that have an arbitrary variable as their p-place argument and, of course, their dominating symbol is f. Thus, not all of the interar's entries contain selection trees, some of them may contain only RPLists or RPLLists; in fact, the RPLLists will be organized as selection trees only when their length exeeds a fixed value. Before starting intersection, we must remember that each resulted u(s)-compatible RP will have to fall into one of the three categories: "pruc", "fwsc" or "bwsc". So we need a way to rapidly recognize to which category it belongs. For this purpose, each leaf of the selection trees contains a "tag" field whose values are: (v, w, cont, pruc, fwsc, bwsc). Now, let's see how and in what context these values are assigned:

- if ds is the pointer to DomSymb structure of t, let id:=ds^.sid and prfp:=ds^.prfp.
- if the argument ti of t is a variable then SearchRP2(prfp^.varg,i,l); put l into interar[i] and tag l with v
- if ti is a constant then SearchRP1(prfp^.const,ti^.cid,i,l1); SearchRP2 (prfp^.varg,i,l2); tag l1 with cont and l2 with w; link l1,l2 and assign the resulted ^RPLList to interar[i],
- if ti is a composite term

```
then SearchRP1(ti^.rp^.cont,id,i,l1)
     if l1 < > nil
       then let  lj1,lj2,lj3  as in (22) section of  Intersect
            procedure;  tag l1 with cont,  lj1 with pruc,  lj2
            with  fwsc  and lj3 with bwsc;  if the  number  of
            these lists is large enough to require  organizing
            them as a selection tree then do so and assign  it
            to interar[i], otherwise link them.
       else suppose {ti^,rp^,pruc} = {rp1,...,rpp1};
                    {ti^.rp^.fwsc} = {rp12,...,rpm2},
                    {ti^.rp^.bwsc} = {rp13,...,rpk3};
            SerchRP1(rp1j^.cont,id,i,lj1), tag lj1 with pruc,
            1 ≤ j ≤ p;
```

Mea 24145

SearchRP1(rp2j^.cont,id,i,1j2), tag 1j2 with fwsc, 1 ⩽ j ⩽ m; SearchRP1(rp3j^.cont,id,i,1j3), tag 1j3 with bwsc, 1 ⩽ j ⩽ k; SearchRP2(prfp^.varg,i,12), tag 12 with w; SearchRP1(ti^.ds^.prfp^.vrp^.cont, id,i,11) tag 11 with cont; organize 111,...,11p, ,121,...,12m,131,...,13k,11,12 as a selection tree (RPLList) and assign it to interar[i].

After these tagging operations are completed for each argument of t, the intersection can begin. If RP is the least element of the updated (ordered) RPLists of interar[i] and this RP was also found in the RPLists of interar[j] (j = i ± 1) then it might be differently tagged. The following rules are used to compute the tag value of a RP taking into consideration the precedent end last tag values of it:

v + v → v
v + w → fwsc
v + cont → cont
v + pruc → pruc
v + fwsc → fwsc
v + bwsc → bwsc

w + w → w
w + cont → fwsc
w + fwsc → fwsc
w + bwsc → pruc
w + pruc → pruc

cont + pruc → pruc
cont + fwsc → fwsc
cont + bwsc → bwsc
cont + cont → cont

fwsc + fwsc → fwsc
fwsc + pruc → pruc
fwsc + bwsc → pruc

bwsc + bwsc → bwsc
bwsc + pruc → pruc

pruc + pruc → pruc

("+" is comutative)

For a practical implementation, the tag values will actually be: 1(v), 2(w), 3(cont), 4(pruc), 5(fwsc), 6(bwsc). A single integer array "tagar" of size [7..42] is maintained; if the selectd RP of interar[i] has the tag value k and the same RP was discovered in interar[j] having the tag value p (1 ⩽ k, p ⩽ 6) then tagar[k + p*6] gives the correct tag value of RP. Therefore, each entry of tagar will contain a number, from 1 to 6, which is given by the above rules. Evidently, this unique tagar is created only once and remains unchanged during the whole computational process. We can observe that if the precedent and last tag values of a RP are identical then the actual tag value will remain the same; also, if the precedent or last tag value is 4(pruc) then 4 remains. With this observation we can minimize the number of times it is necessary to enter into tagar.

When effectively performing the intersection operation, some entries of interar may become empty. If this happens for an entry whose unique RPList is tagged with v(1) then a boolean local variable endfwsc will be set to true signaling that all fwsc type RPs were colected. In this case, the intersection process will continue by completly ignoring every other entry whose RPList is also tagged with v. This is equivalent to jumping over the variable arguments of the term.

Please note that the final tag value of any RP is not v, w nor cont. This is because the RPs corresponding to the set of terms having only variable arguments are not included into the fwsc fields, but instead unique pointers to them are kept in the vrp fields, whereas the cont tag value is imposible to reach for non-integrated terms (as t was assumed).

To end the integration of t, the pointer rp to its rigid part must be added into the following RPLists:

- take {t^.rp^.pruc} = {rp11,...,rpm1} and for each rpj1, 1 ⩽ j ⩽ m, insert rp to its proper place into rpj1^.pruc,

- take {t^. rp^.fwsc} = {rp12,...,rpk2} and for each rpj2, 1 ≤ j ≤ k, insert rp to its proper place into rpj2^.bwsc,
- take {t^.rp^.bwsc} = {rp11,...rpp3} and for each rpj3, 1 ≤ j ≤ p, insert rp to its proper place into rpj3^.fwsc.

If we decide to keep only unique copies of the terms(literals) rigid parts, then a Boyer-Moore ([1]) sort of structure-sharing scheme results and the algorithm for retrieving u(s)-compatible rigid parts works as described, without essential modifications.

We give here the data structures needed to implement the literal indexing mechanism for [1]:

The type TerLit can remain unchanged and is to be used for input objects. The type DomSymb has the fields prp and nrp pointing to ^RPInfo.

```
type RPInfo = record
                rptree : RPTreeAr;
                const : ^ArgRPList1;
                varg : ^ArgRPList2;
                vrp : ^RP
              end;
type RP = record
            rpid : integer;
            rpids : ^IdList;
            lits : ^LitList; {the list of the literals
                              having the respective RP}
            pruc, fwsc, bwsc:^RPList;
            cont:^ArgRPList1
          end;
type LitList = record
                 cl:^Clause;
                 pos:integer;{the position of the literal in
                       the sequence of literals
                       forming the clause given by cl}
                 next:^LitList
               end;
type Clause = record
                .
                .{the seven fields used in [1] to encode a
                . clause}
                lil:^LitInfoList {the informations relate to
                                  the literal list of the
                                  respective clause}
              end;
type LitInfoList = record
                     lit:^TerLit;
                     index:integer;{the input literal and the
                                    index representing the kth
                                    literal in the binding
                                    environment of the
                                    respective clause}
                     rp:^RP; {the pointer to the rigid part
                     of the respective literal}
                     next:^LitInfoList
                   end;
```

3.4 An unification algorithm

In this section, an unification algorithm is presented for the data structures introduced in 3.1.

Let C1,...,Cn be the parent clauses of a resolvent (hyperresolvent) and v:TerLit, v.tltype = 1 a variable. If v occur in Ck then v.clindex[k] will contain the substituent of v and its index. Thus, common variables occurring in distinct clauses can be easely discriminated. Initially, every variable has nil as its substituent and we must retain the list of all variables with the respective indices which were substituted during the unification process. The substituents of these variables must be initialized to nil after the substitution was applied or the unification failed.

Procedure ReturnBinding(v,i,s) returns the last substituent of type Subst for the variable v of index i.

```
procedure ReturnBinding(v:^TerLit; i:integer; var s:Subst);
    var t:^TerLit;
    begin s:=v^.clindex[i]; t:=s.subst;
        while t<>nil do
            begin if t^.tltype=1
                then s:=t^.clindex[s.sindex]; t:=s.subst
                else t:=nil
            end
    end {ReturnBinding}.
```

The function Occur(v,t,i,j) returns true if the variable v of index i occur in the term (literal) t of index j. It is considered that t may have substitued variables which implies the call of ReturnBinding.

```
function Occur(v,t:^TerLit; i, j:integer):boolean;
var tt:^TerLit; targs:^TerLitList; s:Subst;
begin case t^.tltype of
        1: begin ReturnBinding(t,j,s); tt:=s.subst;
            if tt=nil
                then begin if (t=v) and (i=j)
                                then Occur:=true
                                else Occur:=false
                     end
                else Occur:=Occur(v,tt,i,s.sindex)
            end;
        2: begin Occur:=false end;
        3: begin Occur:=false;
                if t^.fpid<>0
                    then targs:=t^.args;
                        while targs<>nil do
                            begin if Occur(v,targs^.terlit,i,j)
                                then Occur:=true;targs:=nil
                                else targs:=targs^.next
                            end
           end
    end {case}
end {Occur}.
```

It is a simple matter to write Occur so that it is called recursively only when the substituent tt is a non-ground composite term ([10d]).

The header of the list of substituted variables will be noted with vSubstVars:^SubstVars, which is a global variable.

```
type SubstVars = record
                   vr:^TerLit;
                   index:integer;
                   next:^SubstVars
                 end.
```

wSubstVars is another global variable which points to a node of vSubstVars list; the procedure DispSubstVars disposes vSubstVars list from its head to wSubstVars (exlusively). This is necessary, for instance, when performing hyperresolution and the k-th negative literal from nucleus failed to unify with a positive literal; in this case, only the variables involved in the last call of the unification algorithm must have their subst field set to nil. So, before attempting to resolve the k-th literal (of nucleus) wSubstVars will be assigned to vSubstVars. If wSubstVars:=nil then DispSubstVars will dispose the whole vSubstVars list.

Procedure Bind(v,t,i,j) binds the term t of index j to the variable v of index i and introduces v into vSubstVars list.

```
procedure Bind (v,t:^TerLit; i,j:integer);
    var sv:^SubstVars;
    begin with v^.clindex[i] do
            begin subst:=t; sindex:=j end;
          sv:=vSubstVars; new(vSubstVars);
          with vSubstVars^ do
            begin vr:=v; index:=i; next:=sv end
    end{Bind}.
```

In seqüel, an unification algorithm is presented which uses a "case" instruction for rapidly detecting the type of terms that have to be unified (avoiding thus a rather cumbersome nesting of "if" statements). A similar version of this algorithm can be found in [10d] which also contain a subsumption algorithm.

```
function Unify(t1,t2:^TerLit;i1,i2:integer):boolean;
var t11,t22:^TerLit; i11,i22:integer; args1,args2:^TerLitList;
s:Subst;
begin if (t1 = t2) and (i1 = i2)
        then Unify:=true
        else begin if t1^.tltype=1
                 then begin ReturnBinding(t1,i1,s);
                      t11:=s.subst;
                      if t11=nil
                        then t11:=t1; i11:=i1
                        else i11:=s.sindex
                      end
               else begin t11:=t1; i11:=i1 end;
             if t2^.tltype=1
              then begin ReturnBinding(t2,i2,s);
                         t22:=s.subst;
                         if t22=nil
                           then t22:=t2; i22:=i2
                           else i22:=s.sindex
                   end
              else begin t22:=t2; i22:=i2 end;
          case ( t11^.tltype + 3 * (t22^.tltype)) of
      {v,v},{v,c} 4,7:begin Bind(t11,t22,i11,i22); Unify:=true end;
        {c,v} 5: begin Bind(t22,t11,i22,i11); Unify:=true end;
        {t,v} 6: begin if Occur(t22,t11,i22,i11)
                       then Unify:=false;DispSubstVars
```

```
                    else Bind(t22,t11,i22,i11); Unify:=true
                end;
    {c,c} 8: begin if t11=t22 then Unify:=true
                              else Unify:=false;DispSubstVars
                end;
{t,c},{c,t} 9,11: begin Unify:=false;DispSubstVars end;
   {v,t} 10: begin if Occur(t11,t22,i11,i22)
                   then Unify:=false; DispSubstVars
                   else Bind(t11,t22,i11,i22); Unify:=true
                end;
    {t,t}12: begin if (t11^.fpid=0) and (t22^.fpid=0)
                   then begin if t11=t22
                              then Unify:=true
                              else Unify:=false;DispSubstVars
                        end
                   else begin if t11^.ds< >t22^.ds
                              then Unify:=false;DispSubstVars
                              else args1:=t11^.args;args2:=t22^.args;
                                   Unify:=true;
                                   while args1< >nil do
                                     begin if Unify(args1^.terlit,
                                                    args2^.terlit,
                                                    i11,i22)
                                           then args1:=args1^.next;
                                                args2:=args2^.next
                                           else Unify:=false;
                                                DispSubstVars;
                                                args1:=nil
                                     end
                        end
            end
        end{case}
    end
end{Unify}.
```

The pairs associated to each label of "case" vizualize the type of terms which are to be unified: v,c,t stands for variables, constants and respectively composite terms(literals).

Let L1 be a literal of rigid part RP1 which is u-compatible with a rigid part RP2. Obviously, each literal L2 of RP2^.terslits list is potentially unifiable with L1. However, Unify(L1,L2,i1,i2) may result to be false. If the unification conflicts were induced exlusively by the flexible part of L1 over the rigid part of L2 (Theorem 4) then the whole RP2^.terslits list of literals can be discarded from the unification with L1.

Example. (a) L1 = P(f(x),h(y,g(x))),L2 = P(f(a),h(f(z),g(b)))
(b) L1 = P(f(x),h(a,g(x))),L2 = P(f(y),h(y,g(b)))

For (a), the unification conflict is induced by the flexible part of L1 over the rigid part of L2 without any "contribution" from the flexible part of L2. For (b), there is L2' a literal whose rigid part is RP2 but which is unifiable with L1 because it has a different flexible part (L2' = P(f(y),h(z,g(b)))).

Because there is no point in persevere to unify the literal L1 with literals L2 (as in (a)) having the same rigid part, a way must be found to reject the literals of RP2^.terslits from the unification with L1. Initially, the unification algorithm was written to incorporate such a mechanism. However, the resulted algorithm performed some superfluous operations in case the respective literals showed to be unifiable. Work is undergoing

for writting the final version of this algorithm. Below a procedure is presented which can be used independently from the unification algorithm and which detects if the flexible part of a term (literal) induces an unification conflict over the rigid part of another term (literal).

```
function UnifTest1(t1,t2:^TerLit):boolean;
  begin UnifTest1:=true;
     case (t1^.tltype + 3*(t2^.tltype)) of
{c,c}8:begin if t1< >t2 then UnifTest1:=false;DispSubstVars end
{t,t}12:begin if t1^.ds = t2^.ds
     then args1:=t1^.args;args2:=t2^.args;
          while args1 < > nil do
               begin if UnifTest1(args1^.terlit,args2^.terlit)
                     then args1:=args1^.next;
                          args2:=args2^.next
                     else UnifTest1:=false;args1:=nil;
                          DispSubstVars
               end
     else UnifTest1:=false;DispSubstVars
     end
   end{case}
end{UnifTest1}.
```

The function UnifTest1 returns true if t1 and t2 have u-compatible rigid parts, and false otherwise;(args1,args2 are global variables).

The mentioned unification conflict is detected by UnifTest(t1,t2,i1,i2), where only the variables of t1 are assumed to be bound from previous unifications. The variables of t1 are permitted to be substituted only directly to constants or composite sub-arguments of t2 without intermediate variables from t2.

```
function UnifTest(t1,t2:^TerLit;i1,i2:integer):boolean;
var t11:^TerLit;i11:integer;args1,args2:^TerLitList;s:Subst;
begin if i1 = i2
     then UnifTest:=UnifTest1(t1,t2)
     else begin UnifTest:=true;
          if t1^.tltype=1
             then begin ReturnBinding(t1,i1,s);
                  t11:=s.subst;
                  if t11=nil then t11:=t1; i11:=i1
                             else i11:=s.sindex
                  end
             else begin t11:=t1; i11:=i1 end;
            case (t11^.tltype + 3*(t2^.tltype)) of
        {v,c},{v,t} 7,10:begin Bind(t11,t2,i11,i2) end;
          {c,c} 8:begin if t11< >t2
                        then UnifTest:=false
                  end;

          {t,t}12:begin if i11< >i2
                        then args1:=t11^.args;args2:=t2^.args;
                             while args1< >nil do
                               begin if UnifTest(args1^.terlit,
                                                 args2^.terlit
                                                 i11,i2)
                                     then args1:=args1^.next;
                                          args2:=args2^.next
                                     else UnifTest:=false;
```

```
                                    args1:=nil;
                                end
                            else UnifTest:=UnifTest1(t1,t2)
                        end
                    end{case};DispSubstVars
                end
            end{UnifTest}.
```

When unifying {t,t} above, if i11 < > i2 then we don't have to check the coincidence of the dominating symbols because this is ensured by the fact that t1 and t2 are u-compatible. However, avoiding this checking into Unify will not actually simplify the algorithm.

As can be easely observed, UnifTest is (in general) much less complex then Unify and it is sufficiently to call UnifTest a single time to see if the flexible part of a literal induces the mentioned type of unification conflict over the rigid part corresponding to a set of literals. Therefore, even if the cardinal of this set is relatively small, the use of UnifTest can be considered as justified when employing the literal indexing filters presented herein and in [3]. For the literal indexing filter of LMA([13], [14]), the UnifTest function is useless because the literals(terms) are not grouped together under the same rigid part criterion, and thus, it is imposible to discard sets of potentially unifiable candidates.

The procedures used to apply the substitutions obtained by Unify and to integrate the resulted objects are presented in [10d] for the structure-sharing scheme of Overbeek et.al.([13],[14]).

4. FINAL REMARKS

Each object is contained in a number of multi-level FPA lists [13] which equals the number of symbols' occurrences of this object. Moreover, the unification properties of [14a] which are attached to every object, do require their own storage. These observations have determined the authors of [3] to conclude that although there is a considerable wasted space in an outline string, this might not be disproportionately (in comparison with the storage required by FPA lists) when maintaining the outlines and the lists of compatible outlines. Now, that the storage inconvenients due to the keeping of outline strings have been completely eliminated, it seems that the literal indexing mechanism presented in this work is the least space consuming.

Referring to the speed of retreiving potentially unifiable candidates, the method of [13] and the one proposed herein are both essentially based on performing intersection of unions of ordered lists. However, there are some reasons to believe that the operations involved in the second method are less complex:
- it is generally more difficult to access a FPA list corresponding to an unification property then to access a (super-)rigid part,
- it is necessary to perform a number of FPA lists intersections which is equal with the number of rigid symbols occurrences of an object to obtain the set of all objects having the same rigid part; for this motive, the number of intersections and the length of the RP-lists involved is smaller,
- we don't obtain a single potentially unifiable candidate, when performing an intersection, but generally, a set of potential

candidates and this is obviously important.

Finally, there is no claim that the literal indexing mechanism included herein is more efficient then the one presented in [13]; only an experimental comparison could prove or disprove this.

ACNOWLEDGEMENT

I want to express my gratitude to Dr. Lawrence Henschen for emphasizing some aspects concerning his work on formula outline. I am grateful to Dr. Alexander Herold and Dr. Hans Jurgen Ohlbach for their advices and for sending to me some useful works related to literal indexing mechanisms. Many thanks are addressed to my collegues Rolanda Predescu, Anca Hotaran and Octavian Scortea.

REFERENCES

[1] Boyer,R.S., Moore,J.S. The Sharing of Structure in Theorem-Proving Programs. In Machine Intelligence,vol.7,1972.

[2] Chang,C.L., Lee,C.T. Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York, 1973.

[3] Henschen,L.I., Naqvi,S.A. An Improved Filter for Literal Indexing in Resolution Systems. In Proc. of the 7th IJCAI, Vancouver, 1981.

[4] Henschen,L.I., Overbeek,R., Wos,L. Hyperparamodulation: A refinement of paramodulation. In Proc. of the 5th CADE, Springer-Verlag, LNCS vol.87, 1980.

[5] Herold,A. Some Basic Notions of First-Order Unification Theory. Universitat Karlsruhe, Interner Bericht nr. 15/83, 1983.

[6] Knuth,D.E. The Art of Computer Programming, vol.3. Addison-Wesley, Reading, Mass. 1973.

[7] Lankford,D.S., Ballantyne,A.M. The Refutation Completeness of Blocked Permutative Narrowing and Resolution. In Proc. of 4th CADE, Austin, TX, 1979.

[8] Loveland,D.W. Automated Theorem Proving: A Logical Basis. North-Holland, 1978.

[9] Mark Karl G Ralph. The Markgraf Karl Refutation Procedure. Memo-SEKI-MK-84-01, 1984.

[10] Nicolaita,D.(reports written in romanian):

(a) Resolution Automated Theorem-Proving Methods. Report I.T.C.I.,AI-FD-047,1984. Institute for Computers and Informatics,Bucharest.

(b) Paramodulation Methods in Automated Deduction. Report I.T.C.I,AI-FD-60,1985. Institute for Computers and Informatics, Bucharest.

(c) A Study Concerning the Realisation of an Automated Theorem-Prover I(theoretical aspects). Technical Report, 1985. National Institute for Scientific and Technical Creation, Bucharest.

(d) A Study Concerning the Realisation of an Automated Thorem-Prover II(implementation aspects) Technical Report, 1986, National Institute for Scientific and Technical Creation, Bucharest.

(e) Implementation Techniques for Resolution Systems. In Proc. of INFOTEC, Bucharest 1986.

[11] Ohlbach,H.I.
(a) Terminator. In Proc. of 8th IJCAI, Karlsruhe 1983.
(b) Theory Unification in Abstract Clause Graphs. Memo-SEKI-85-I-KL, 1985.

[12] Overbeek,R. An Implementation of Hyperresolution. In Comp. & Maths with Appls.,vol.1,1975.

[13] Ovebeek,R.,Lusk,E. Data Structures and Control Architecture for the Implementation of Theorem-Proving Programs. In Proc. of the 5th CADE, Springer-Verlag, LNCS vol.87,1980.

[14] Overbeek,R.,Lusk,E.,McCune,W.
(a) Logic Machine Architecture: Kernel Function and Inference Mechanisms. In Proc. of the 6th CADE, Springer-Verlag, LNCS vol.138,1982.
(b) ITP at Argonne National Laboratory, Paths to High-Performance Automated Theorem Proving. In Proc. of the 8th CADE, Springer-Verlag, LNCS vol.230,1986.

[15] Ovebeek,R.,McCharen,I.,Wos,L. Complexity and Related Enhancements for Automated Theorem Proving Programs. In Comp.&Maths with Appls, vol.2,1976.

[16] Siekmann,I.H. Univerasal Unification. In Proc of the 7th CADE, Springer-Verlag, LNCS vol.170, 1984.

[17] Stickel,M.E. Automated Deduction by Theory Resolution. In J. of Automated Reasoning, vol.1,1985.

[18] Veroff,R. Canonicalization and Demodulation. Report ANL-81-6, Argonne National Laboratory, 1981.

[19] Walther,C. A Many-Sorted Calculus Based on Resolution and Paramodulation. Memo-SEKI 34-82, 1982. Universitat Karlsruhe.

[20] Winker,S. An Evaluation of an Implementation of Qualified Hyperresolution. In IEEE Trans. on Comp.,C-25(8) 1976.

[21] Wos,L.,McCune,W. Negative Paramodulation. In Proc. of the 8th CADE, Springer-Verlag, LNCS vol.230, 1986.

[22] Wos,L.,Robinson,G.A.,Carson,D.,Shalla,L. The Concept of Demodulation in Theorem Proving. In JACM vol.14(4), 1967.