# Combining Rank/Activity with Set of Support, Hyperresolution and Subsumption

J. D. Horton and Bruce Spencer

Faculty of Computer Science, University of New Brunswick
P.O. Box 4400, Fredericton, New Brunswick, Canada E3B 5A3
*jdh@unb.ca, bspencer@unb.ca, http://www.cs.unb.ca*

**Abstract.** The rank/activity (R/A) restriction of binary resolution specifies that literals in clauses have ranks, and are either active (can be resolved) or inactive. When a literal is resolved, literals in the new clause, that were of lower rank in the parent clause than the resolved literal, become inactive in the child clause. Literals that merge or factor with literals from other clauses become active. Two derivations are equivalent if they consist of the same resolutions, possibly in a different order, but resulting in the same clause. It is known that with R/A, no two equivalent derivations are found (uniqueness), but that one derivation from each equivalence class is found. In this paper, R/A is combined with the set of support strategy, with hyperresolution, and also with subsumption. In the first two cases, both completeness and uniqueness are maintained. In the subsumption case, completeness and uniqueness can be retained if full rank/activity is kept, but subsumption of small clauses by bigger clauses is not done. With full subsumption, and a restricted version of R/A, completeness is kept but uniqueness is lost.

## 1 Introduction

Resolution proof procedures start with a set of clauses corresponding to the negation of the expression to be proved, and then resolve pairs (hyperresolving sets) of clauses to form new clauses until the empty clause is found. If the parent clauses are recorded, then the derivation of each clause can be readily found. Thus one can think of a proof procedure as searching through the space of all derivations of all clauses starting from the set of input clauses.

Derivations can be represented in several ways: as a sequence of clauses, as a proof tree which we call a binary resolution tree[8], or as a clause tree[3]. Simply reordering resolutions does not change the essence of the derivation, but this not readily detectable when the derivation is just a sequence of clauses. The clauses in two equivalent derivations may not even be the same. However in binary resolution trees and clause trees, it is readily detectable. Two binary resolution trees are defined to be rotation equivalent if there is a reordering of the resolutions that transforms one tree to the other. Two clause trees are said to be reversal equivalent if there is a sequence of path reversals that transform one to the other. These two concepts, rotation equivalence classes of binary resolution trees and reversal equivalence classes of clause trees, both partition the set of derivations into the same equivalence classes, in which derivations are considered to be equivalent if the same resolutions are performed in both.

The simplest restriction, which is assumed through the remainder of the paper, is to disallow the same resolution twice. If $C_1$ is resolved with $C_2$, then $C_2$ will not later be resolved with $C_1$.

Subsumption, which rejects clauses that are subsumed by other clauses, removes a large portion of the search space itself, but clearly retains completeness as there always exist derivations equally good or better than any derivation that has been removed. Regularity [9], and its generalization minimality [8], prevent some redundant refutations from being produced but always leave some smaller refutations. The set of support strategy (SOS) prevents the resolution of two clauses outside the set of support [5]. Hyperresolution allows only positive clauses to be produced, forcing all negative literals to be resolved at once. Many, but not all, refutations cannot be reached by any of these restrictions. All the above restrictions allow some refutations to be constructed many times.

The major focus of this paper is the rank/activity restriction (R/A) introduced in [4], and its interaction with the various restrictions mentioned above. In the R/A restriction, the literals in each clause are ordered by a rank function which assigns an integer value to each literal. This rank function must be consistent between a parent clause and a child clause, in that if $rank(a) < rank(b)$ in a parent clause, then $rank(a) < rank(b)$ in the child clause as well. When a clause is resolved on a literal of a given rank, in the newly created child clause all literals of a lesser rank (rank as defined in the parent clause) are deactivated, and hence are not allowed to be resolved again. This activity condition also must be inherited from parent literal to child literal, with the following exception. When two literals that come from different parents merge (or factor) in a child clause, the literal becomes active, regardless of whether the literals in the parent clauses are active or inactive. This exception is very important as completeness is lost if this is not done. An implementation need not actually factor unifiable literals immediately, but just assign a rank to the set of unifiable literals. The rank can be anything, but one possibility is to make the rank of the set equal to the rank of the lowest ranked literal.

In contrast to all the other restriction mentioned above, no equivalence class of refutations is removed by R/A (strong completeness), but only one refutation per class is found (uniqueness). We investigate how R/A interacts with the above restrictions. We already know that R/A and minimality combined is both complete and unique [4]. R/A, slightly weakened, combined with SOS is shown to be complete and unique in section 3. Another slightly weakened form of R/A combined with hyperresolution is also complete and unique, as shown in section 4. R/A and subsumption is investigated in section 5: full R/A with a much weakened subsumption is shown to be both complete and unique; a weakened form of R/A and full subsumption is complete but not unique.

## 2 Background

We use standard definitions [1] for atom, literal, substitution, unifier and most general unifier. Much of this section originated from [8]. In the following a *clause* is an unordered disjunction of literals. An atom $a$ *occurs in* a clause $C$ if either $a$ or $\neg a$ is one of the disjuncts of the clause. The clause $C$ *subsumes* the clause $D$ if there exists a substitution $\theta$ such that $C\theta \subseteq D$. Two clauses are *standardized apart* if no variable occurs in both. Given two *parent* clauses $C_1 \vee a_1 \vee \ldots \vee a_m$ and $C_2 \vee \neg b_1 \vee \ldots \vee \neg b_n$ which are standardized apart (a variable renaming substitution may be required) their *resolvent* is the clause $(C_1 \vee C_2)\theta$ where $\theta$ is the most general unifier of $\{a_1, \ldots, a_m, b_1, \ldots, b_m\}$. The *atom resolved upon* is $a_1\theta$, and the set of *resolved literals* is $\{a_1, \ldots, a_m, \neg b_1, \ldots, \neg b_m\}$. An implementation is free to merge or factor literals if desired. Factoring may be seen as an optimization if the factored clause can be used in several resolution steps, since the factoring is done only once.

A *fair* resolution procedure is does every resolution not otherwise restricted, and does it exactly once. No resolution is deferred forever. Thus if no clauses are rejected, a fair resolution procedure produces every possible derivation.

### 2.1 Binary resolution tree definitions

A binary resolution derivation is represented by a binary tree, drawn with its root at the bottom. Each edge joins a *parent* node, drawn above the edge, to a *child* node, drawn below it. The *ancestors* (*descendants*) of a node are defined by the reflexive, transitive closure of the parent (child) relation. The *proper ancestors* (*proper descendants*) of a node are those ancestors (descendants) not equal to the node itself. Thus the root is a descendant of every node in the tree.

**Definition 1.** *A* binary resolution tree *on a set $S$ of input clauses is a labeled binary tree. Each node $N$ in the tree is labeled by a* clause label, *denoted $cl(N)$. Each node either has two parents and then its clause label is the result of a resolution operation on the clause labels of the parents, or has no parents and is labeled by an instance of an input clause from $S$. In the case of a resolution, the atom resolved upon is used as another label of the node: the atom label, denoted $al(N)$. Any substitution generated by resolution is applied to all labels of the tree. The clause label of the root of the binary resolution tree is called the* result *of the tree, $result(T)$. A binary resolution tree is* closed *if its result is the empty clause, $\square$.*

For the binary resolution tree in Figure 1 $S = \{a \vee d, \neg a \vee b \vee \neg e, c \vee \neg d, e \vee f \vee g, a \vee b \vee \neg c, \neg a \vee h, \neg h, \neg b, \neg g\}$. The labels of a node $N$ are displayed beside the name of the node and separated by a colon if both labels exist. For example the node $N_4$ has atom label $c$, and clause label $a \vee b \vee b \vee f \vee g$.

We can trace what happens to a literal from its occurrence in the clause label of some leaf, down through the tree until it is resolved away. Clearly if all literals are eventually resolved away, the clause label of the root is empty. In this case by

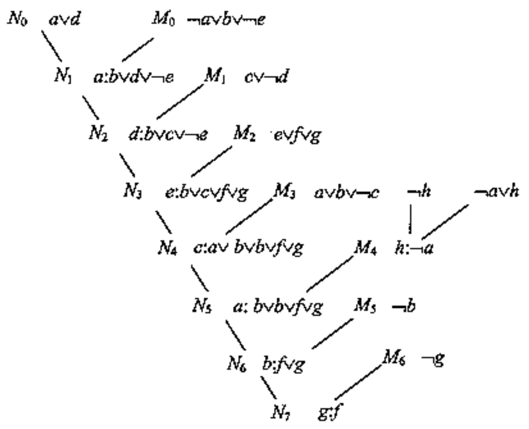
$N_0$ $a \vee d$
$M_0$ $\neg a \vee b \vee \neg e$
$N_1$ $a{:}b \vee d \vee \neg e$
$M_1$ $c \vee \neg d$
$N_2$ $d{:}b \vee c \vee \neg e$
$M_2$ $e \vee f \vee g$
$N_3$ $e{:}b \vee c \vee f \vee g$
$M_3$ $a \vee b \vee \neg c$
$\neg h$
$\neg a \vee h$
$N_4$ $c{:}a \vee b \vee b \vee f \vee g$
$M_4$ $h{:}\neg a$
$N_5$ $a{:}b \vee b \vee f \vee g$
$M_5$ $\neg b$
$N_6$ $b{:}f \vee g$
$M_6$ $\neg g$
$N_7$ $g{:}f$

**Fig. 1.** A binary resolution tree.

soundness of resolution, the clause labels of the leaves is an unsatisfiable set of clauses. Thus we are primarily concerned about tracing the "history" of a literal starting from its appearance in a leaf.

For example in Figure 1, $(M_1, N_2, N_3)$ is a history path for $c$ which closes at $N_4$. The two history paths for $b$ in Figure 1, corresponding to the two occurrences of $b$, are $(M_3, N_4, N_5)$ and $(M_0, N_1, N_2, N_3, N_4, N_5)$. Both of these close at $N_6$. The only history path which does not close is the one for $f$, which is $(M_2, N_3, N_4, N_5, N_6, N_7)$.

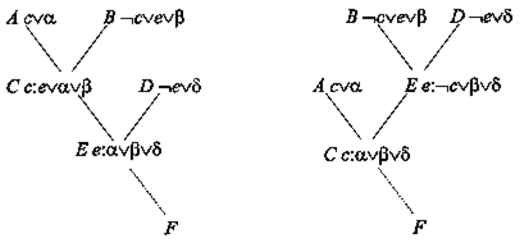
$A$ $c \vee \alpha$
$B$ $\neg c \vee e \vee \beta$
$C$ $c{:}e \vee \alpha \vee \beta$
$D$ $\neg e \vee \delta$
$E$ $e{:}\alpha \vee \beta \vee \delta$
$F$
$B$ $\neg c \vee e \vee \beta$
$D$ $\neg e \vee \delta$
$A$ $c \vee \alpha$
$E$ $e{:}\neg c \vee \beta \vee \delta$
$C$ $c{:}\alpha \vee \beta \vee \delta$
$F$

**Fig. 2.** A binary tree rotation

**Operation 1 (Edge Rotation)** *Let $T$ be a binary resolution tree with an edge $(C, E)$ between internal nodes such that $C$ is the parent of $E$ and $C$ has two parents $A$ and $B$. Further, suppose that no history path through $A$ closes at $E$. Then the result of a* rotation *on this edge is the binary resolution tree $T'$ defined by resolving $cl(B)$ and $cl(D)$ on $al(E)$ giving $cl(E)$ in $T'$ and then resolving*

*$cl(E)$ with $cl(A)$ on $al(C)$ giving $cl(C)$ in $T'$. Any history path closed at $C$ in $T$ is closed at $C$ in $T'$; similarly any history path closed at $E$ in $T$ is closed at $E$ in $T'$. Also, the child of $E$ in $T$, if it exists, is the child of $C$ in $T'$. (See Figure 2).*

A rotation may introduce tautologies to clause labels of internal nodes. For instance, if $al(C)$ occurs in $cl(D)$ then $cl(E)$ in $T'$ may be tautological. However the clause label of the root is not changed (Corollary 1).

**Corollary 1.** *Given a binary resolution tree $T$ with an internal node $C$ and its child $E$, the rotation of edge $(C, E)$, Operation 1, generates a new binary resolution tree and $cl(C) = cl(E)$ up to variable renaming.*

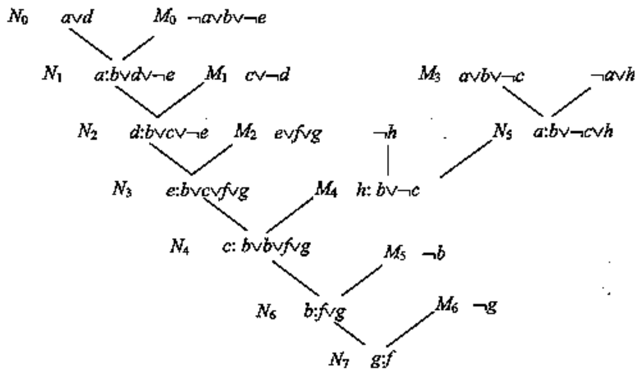
$N_0$ $a \vee d$
$M_0$ $\neg a \vee b \vee \neg e$
$N_1$ $a{:}b \vee d \vee \neg e$
$M_1$ $c \vee \neg d$
$M_3$ $a \vee b \vee \neg c$
$\neg a \vee h$
$N_2$ $d{:}b \vee c \vee \neg e$
$M_2$ $e \vee f \vee g$
$\neg h$
$N_5$ $a{:}b \vee \neg c \vee h$
$N_3$ $e{:}b \vee c \vee f \vee g$
$M_4$ $h{:}\ b \vee \neg c$
$N_4$ $c{:}\ b \vee b \vee f \vee g$
$M_5$ $\neg b$
$N_6$ $b{:}f \vee g$
$M_6$ $\neg g$
$N_7$ $g{:}f$

**Fig. 3.** From Figure 1 rotate $(N_4, N_5)$, then $(M_4, N_5)$

A rotation changes the order of two resolutions in the tree. Rotations are invertible; after a rotation, no history path through $D$ closes at $C$, so another rotation at $(E, C)$ can be done, which generates the original tree again. We say that two binary resolution trees are *rotation equivalent* if one can be generated from the other by a sequence of rotations. For instance, the binary resolution tree in Figure 1 is rotation equivalent to the tree in Figure 3, which is produced by rotating the edge $(N_4, N_5)$ in Figure 1 and then the edge $(M_4, N_5)$. Rotation equivalence is an equivalence relation.

A rank function must assign a value to every literal in the clause at each node in a given binary resolution tree, in such a way that it orders history paths consistently. Moreover it must assign values to sets of literals if they are unified by a resolution closer to the root of the binary resolution tree. In the following definition a rank function is required to assign values to every set of unifiable literals, even if they are not unified until later in the tree. The rank of a set of literals could be given as the minimum of the ranks of the literals unified, the maximum, or any other number. In the following, if $H$ is set of history paths in a binary resolution tree $T$ with unifiable literals and they have some node in common, let $literal(H)$ be the multiset of these literals.

**Definition 2 (Rank function).** *Let $\mathcal{F}$ be a set of binary resolution trees, closed under taking subtrees. Let $r$ assign a value to every set of unifiable literals at every node of every tree. Then $r$ is a* **rank function** *for $\mathcal{F}$ if $r$ satisfies the following condition in every binary resolution tree $T$:*

*For every pair of disjoint sets $H_1$ and $H_2$ of history paths which have two nodes $N_1$ and $N_2$ in common:*

$$r(literal(H_1), N_1) < r(literal(H_2), N_1) \iff$$
$$r(literal(H_1), N_2) < r(literal(H_2), N_2).$$

Thus $r$ is a rank function if it orders the sets of history paths consistently. In fact the reflexive transitive closure of this relation between sets of history paths, is a partial order.

Next we want to define those binary resolution trees which can be built using the R/A restriction. Let $N$ be a node other than the root of a binary resolution tree $T$. Let $H(N)$ be the set of history paths with $N$ as their head. Then these paths close at the child of $N$.

**Definition 3 (r-compliant).** *Let $r$ be a rank function for a binary resolution tree and all its subtrees. Then $T$ is* **$r$-compliant** *if the following condition is true: Let $N$ and $M$ be any two nodes such that $H(N)$ also have $M$ in common. Thus $M$ is an ancestor of $N$. Then $r(literal(H(M)), M) \leq r(literal(H(N)), M)$.*

The resolution at $M$'s child does not deactivate the set of history paths with head at $N$. Moreover, this set of history paths is not affected by what happens before they are drawn together at some node by a resolution. Therefore it is created as an active set of literals. Hence the set is active in $N$ and can be resolved by a R/A procedure at $N$'s child. Thus the $r$-compliant binary resolution trees are precisely those trees which can be constructed using the R/A restriction of binary resolution, using the function $r$ as the rank function.

## 2.2 Clause tree definitions

The definition of clause tree in this paper differs from that in [3]. There the definition is procedural, in that operations that construct clause trees are given. Here the definition is structural.

**Definition 4 (Clause Tree).** *$T = \langle N, E, L, M\rangle$ is a clause tree on a set $S$ of input clauses if*

1. *$\langle N, E\rangle$ is an unrooted tree.*
2. *$L$ is a labeling of the nodes and edges of the tree. $L : N \cup E \rightarrow S \cup A \cup \{+, -\}$, where $A$ is the set of instances of atoms in $S$. Each node is labeled either by a clause in $S$ and called a clause node, or by an atom in $A$ and called an atom node. Each edge is labeled $+$ or $-$.*

3. *No atom node is incident with two edges labeled the same.*
4. *Each edge $e = \{a, c\}$ joins an atom node $a$ and a clause node $c$; it is associated with the literal $L(e)L(a)$.*
5. *For each clause node $c$, $\{L(a,c)L(a) | \{a,c\} \in E\}$ is an instance of $L(c)$. A path $\langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ where $0 \leq i \leq n$, $v_i \in N$ and $e_j \in E$ where $1 < j < n$ is a* merge path *if $L(e_1)L(v_0) = L(e_n)L(v_n)$. Path $\langle v_0, \ldots, v_n\rangle$* precedes *($\prec$) path $\langle w_0, \ldots, w_m\rangle$ if $v_n = w_i$ for some $i = 1, \ldots, m-1$*
6. *$M$ is the set of merge paths called chosen merge paths such that:*
   *(a) the tail of each is a leaf (called a* closed *leaf),*
   *(b) the tails are all distinct and different from the heads, and*
   *(c) the relation $\prec$ on $M$ can be extended to a partial order.*

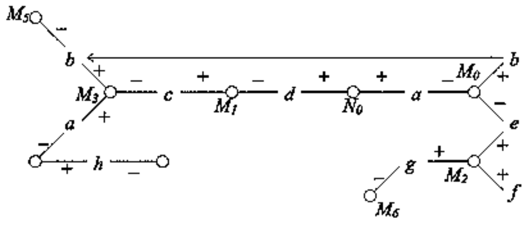

**Fig. 4.** A clause tree corresponding to the binary resolution trees in Figures 1 and 3.

In this paper we disallow merge paths of length two since they correspond to factoring an input clause, which can be the clause of a clause node. A set $M$ of paths in a clause tree is *legal* if the $\prec$ relation on $M$ can be extended to a partial order. A path $P$ is *legal in* $T = \langle N, E, L, M\rangle$ if $M \cup \{P\}$ is legal. If the path joining $t$ to $h$ is legal in $T$, we say that $h$ is *visible* from $t$. A path $\langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ where $v_i \in N$ and $e_j \in E$ is a *tautology path* if $L(v_0) = L(v_n)$ and $L(e_1) \neq L(e_n)$. A path is a *unifiable tautology path* if $L(e_1) \neq L(e_n)$ and there exists a substitution $\theta$ such that $L(v_0)\theta = L(v_n)\theta$. A path is a *unifiable merge path* if there exists a substitution $\theta$ such that $L(e_1)L(v_0)\theta = L(e_n)L(v_n)\theta$.

A clause tree with a single clause node is said to be *elementary*. An *open leaf* is an atom node leaf that is not the tail of any chosen merge path. The disjunction of the literals at the open leaves of a clause tree $T$ is called the *clause* of $T$, $cl(T)$.

**Definition 5 (Minimal clause tree).** *A clause tree $\langle N, E, L, M\rangle$ is* minimal *if it contains no legal merge path not in $M$ and no legal tautology path.*

Operations that can be applied to clauses include:

1. Construction of elementary clause trees from (an instance of) an input clause with one clause node and one atom node leaf for each literal in the clause.
2. Resolution of two clause trees by identifying the open leafs, one from each clause tree, of the literals to be resolved.

3. Choosing a merge path between two open leaves that correspond to the same literal.
4. Taking an instance of a clause tree by applying a substitution to each of the atom labels.

In this paper, taking instances and choosing merge paths are only done immediately before a resolution.

## 3 Set of Support with Rank/Activity

The SOS strategy, or restriction, requires that the unsatisfiable set $S$ of clauses be partitioned into two parts: a set of support $S'$ and a set $S'' = S - S'$ that is satisfiable. It restricts pairs of clauses from $S''$ from resolving. An SOS binary resolution tree is one in which each internal node has at least one leaf ancestor from the set of support. It is well known that the SOS restriction preserves completeness [5] and so an SOS binary resolution tree refutation must exist.

Rank/activity needs to be slightly weakened to be combined with SOS. When a clause from $S''$ is resolved (with a clause from the set of support) the literals from it are all active in the resulting clause. Thereafter the R/A restriction is applied as usual. In effect, these literals are not ranked until their clause is resolved with a clause from the set of support.

**Theorem 2.** *Let $S$ be an unsatisfiable set of clauses and $S'$ be a subset of $S$ such that $S - S'$ is satisfiable. Let $r$ be a rank function of the binary resolution trees of $S$. There exists an $r$-compliant SOS binary resolution tree refutation on $S$ with set of support $S'$.*

*Proof.* Let $T$ be a binary resolution tree on $S$ with set of support $S'$. We construct $T^*$ that is rotation equivalent to $T$ and $r$-compliant. Assume $T$ has three or more nodes, since a one node tree is $r$-compliant.

For each leaf $L$ define an internal node $P(L)$ that $L$ points to. If $L$ is labeled with a clause from $S - S'$, let $L$ point to its child. Otherwise $L$ is labeled with a clause from $S'$ and the node it points to is assigned as follows: Consider the descendants $D_1, ..., D_k$ of $L$ that resolve away only literals from $L$. Thus a descendant $D_i$ is in this set if for each history path $H$ in $H(D_i)$, $tail(H) = L$. These descendants are not merged with literals from other leaves before they are resolved. There must be at least one of these. Now consider that descendant $D_j$ of $L$ such that $r(literal(H(D_j)), L)$ is minimal over $j = 1, ..., k$. The child of $D_j$ resolves the lowest ranked set of literals in $L$. Let $L$ point at the child of $D_j$.

Because there is one more leaf than internal nodes, some internal node $E$ is pointed at by (at least) two leaves $L_1$ and $L_2$. (In fact there must be exactly two.) If both leaves are not from the SOS then $T$ is not an SOS binary resolution tree, so at least one of these leaves must be from the SOS. Construct $T_0$, rotation equivalent to $T$, but where $E$ is the child of two leaves. If this is not already true, without loss of generality let $L_1$ be the leaf ancestor from the set of support, and let $B$ and $C$ be the grandparent and parent, respectively, of $E$, that are descendants of $L$. Note that the SOS leaf ancestor must be at least two levels from $E$ because a non-SOS leaf ancestor of $E$ is a parent of $E$. A rotation of the edge $(C, E)$ must be possible because there are no merges on literals resolved at $E$. After the rotation $L_1$ and $L_2$ both still point to $E$ and $E$ has fewer ancestors. $T_0$ is constructed after a finite number of such rotations.

From $T_0$ construct a smaller tree $T_1$ by removing $L_1$ and $L_2$, making $E$ a new leaf. Note that $r$ is still a rank function on $T_1$. By induction $T_1$, which has fewer nodes, is rotation equivalent to a binary resolution tree $T_1'$ that is $r$-compliant. Construct $T'$ from $T$ by replacing the leaf nodes $L_1$ and $L_2$ above $E$. $T'$ is rotation equivalent to $T_0$, as is seen by performing on $T_0$ those rotations performed on $T_1$. Thus $T'$ is rotation equivalent to $T$.

It remains to show that $T'$ is $r$-compliant. The only nodes that must be checked are the new nodes $L_1$ and $L_2$, as a possible $M$ in the definition of $r$-compliant. Consider any non-root node $N'$ such that the history paths $H(N')$ have $L_1$ in common. Then $r(literal(H(L_1)), L_1) \leq r(literal(H(N')), L_1)$ by the definition of $P(L_1)$, so that the $r$-compliant condition is always satisfied at $L_1$. The same situation applies at $L_2$. Thus $T'$ is $r$-compliant. This proves completeness.

If the condition is added that the rank function $r$ does not map two disjoint sets of literals at any node to the same value, then the pointer function chooses a unique node $P(L)$ for any leaf $L$. Let $L$ and $D_i$ be as defined above. Let $T^*$ be any binary resolution tree that is $r$-compliant and rotation equivalent to $T$. The history paths $H(L)$ close at the child of $L$. Then $r(literal(H(L)), L) \leq r(literal(H(D_i)), L)$ for $i = 1, 2, ..., k$, because $T^*$ is $r$-compliant. But by the uniqueness condition on $r$, these ranks must all be distinct. Thus there is a unique $j$ such that $H(L) = H(D_j)$, and $P(L)$ is the child of $D_j$, by the definition of $P(L)$. Hence $D_j = L$, and $P(L)$ must be the child of $L$ in $T^*$.

This argument shows that each leaf of $T^*$ has a unique child. The argument can be extended to all the nodes of $T^*$, by inducting on the height of the subtree above the node. Thus every node, other than the root node, has a uniquely defined child node. It follows that the binary resolution tree $T^*$ is unique. □

**Corollary 2.** *Any fair SOS procedure, when combined with R/A, will generate exactly one binary resolution tree from each class of rotation equivalent SOS binary resolution trees.*

## 4 Hyperresolution with Rank/Activity

One of the most common restrictions of resolution is hyperresolution[7]. It is used by many bottom-up resolution theorem provers, like Otter[6] and Blitzen (de Nivelle). Hyperresolution can be thought of as a restriction of SOS, with the clauses that have all positive literals being in the set of support, and with the other clauses (negative and mixed) forming a satisfiable set of clauses; the empty interpretation is a model. But hyperresolution also adds the restriction that all the negative literals must be resolved at the same time. The result will always be a positive clause so that the clauses in the set of support are always positive. Thus factoring and merging, other than on the input clauses, must occur only with positive literals. Hence if a hyperresolution proof is represented as a clause tree, all the merge paths join positive literals. (The converse of this follows from the completeness theorem below.) We mention clause trees because the completeness and uniqueness theorems below appear to easier to prove using clause trees than using other proof representations.

Rank/activity and hyperresolution can be combined in a way that is similar to combining R/A and SOS. The literals of the negative and mixed clauses are not ranked and remain active until the clause is hyperresolved. (de Nivelle [2] suggested this.) Once the clause has been hyperresolved, the set of unifiable positive literals from it in the child clause become active and are given ranks, which can be done in any arbitrary fashion. The activity and ranks of literals in the positive clauses are manipulated just as in the pure R/A restriction.

Fairness is again an issue. If any hyperresolution becomes possible, and all the positive literals are active, then the procedure must eventually perform the hyperresolution.

**Theorem 3.** *Any fair R/A hyperresolution procedure, when applied to a set $S$ of clauses, will generate any clause tree $T$ defined on $S$ that has only positive merge paths. Moreover, $T$ is produced exactly once by the procedure.*

*Proof.* Consider and clause tree $T$ on a set $S$ of input clauses with a rank function $r$. The proof is by induction on the number of nodes in $T$. The induction hypothesis is strengthened to assume only that those literals that are not the head edges of merge path in $T$ are active.

Consider the tree $T'$ derived from $T$ which has as it nodes the clause nodes of $T$. Two (clause) nodes are connected in $T'$ if there is an atom node in $T$ that is adjacent to both. (See Figure 5.)

Consider each edge $\{v, w\}$ of $T'$. Let $b$ be the atom node in $T$ between $v$ and $w$. If $(w, b)$ is labeled negatively, and hence $(v, b)$ is positive, then direct the edge in $T'$ from $w$ to $v$. In this way direct all edges of $T'$, except for one edge at each node that corresponds to a positive input clause. Let $C$ be the (possibly factored) positive input clause of one of these positive clause nodes $u$. Consider the set of nodes adjacent to $u$ in $T$ that are not the head or tail of any merge path over $u$. Let $a$ be the minimally ranked atom label among these atom nodes. Direct the edge in $T'$ corresponding to $a$ away from $u$. (See Figure 5.)

Since $T'$ is a directed acyclic graph, it must have a sink, $v$, at which all incident edges are directed. The node $v$ cannot correspond to a positive clause, because all positive clauses have one edge directed away. Therefore $v$ must correspond to a negative or mixed clause. A hyperresolution can be performed involving that clause using its negative literals, and those neighbouring clause nodes labelled with positive clauses. In Figure 5, $u$ and $x$ are examples of these. Note that this hyperresolution can deactivate a literal in such a positive clause only if the literal is the head or tail of a merge path over that clause node.

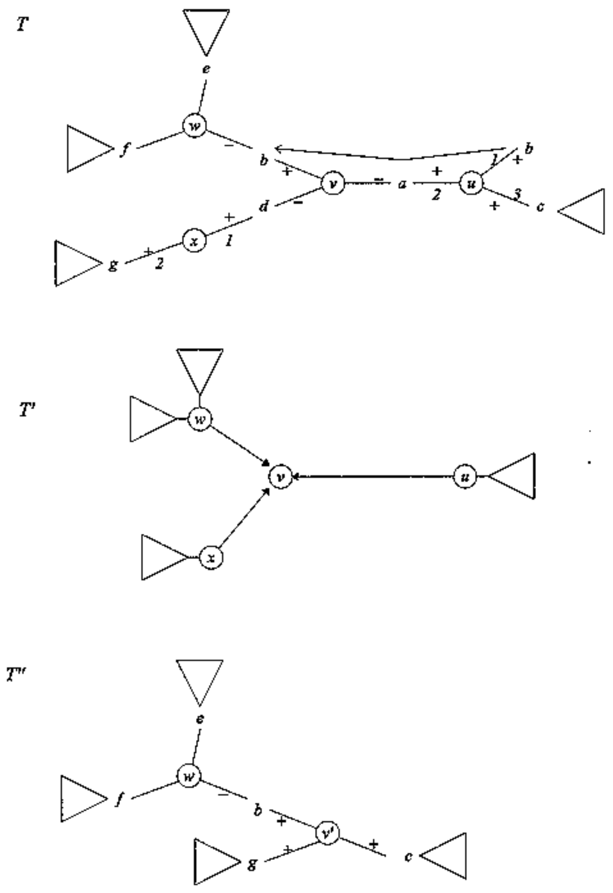

**Fig. 5.** An example of the construction in Theorem 3

Now consider the clause tree $T''$ corresponding to the tree derived from $T$ by contracting all edges adjacent to $v$, to produce a new (clause) node $v'$. This action is equivalent to performing the hyperresolution indicated above and using the result as a new input clause. Any merge path in $T''$ that includes only one clause node, which can only be $v'$, must be removed, with the path's tail edge and adjacent closed leaf being removed as well. Such a merge path is shown in Figure 5 between $b$'s. By the induction hypothesis, the clause tree $T''$ is constructed by any fair R/A hyperresolution procedure, starting with $S \cup \{clause(u)\}$.

We must now specify what procedure is to be used on $S \cup \{clause(u)\}$. Choose the same procedure, that is sequence of hyperresolutions, that is done starting from $S$, except the hyperresolution that creates $\{clause(u)\}$. This procedure is fair because the original procedure is fair. By the induction hypothesis, $T''$ must be constructed. Thus $T$ is constructed by the original procedure. Hence R/A combined with hyperresolution is complete; all clause trees that can be produced by hyperresolution are produced by any fair R/A hyperresolution procedure.

A similar induction proof shows uniqueness. By an induction hypothesis, the clause tree corresponding to $T''$ is constructed in only one way by the procedure. There is only one construction of $T$ by hyperresolution with input clauses. In any other construction of $T$, one of the positive clauses $C$ that corresponds to a neighbour of $v$ in $T'$, must be used in some other hyperresolution. But then the literal of $C$ resolved is ranked higher than the literal that connects the node of $C$ to $v$. Then this latter literal becomes inactive, and can never be resolved with the complementary literal of the clause of $v$. For example in Figure 5 consider $clause(x) = \{d, g\}$. If $g$ were resolved before $d$, then $d$ would become inactive, and could not resolve with the $\neg d$ of $clause(v)$. Thus $T$ can only occur once in the sequence of clause trees produced by the R/A hyperresolution procedure. □

**Corollary 3.** *A clause tree $T$ can be constructed by hyperresolution if and only if $T$ has no merge paths in negative literals.*

The minimality restriction cannot be combined in total with hyperresolution. Negative merge paths cannot be chosen, but unchosen ones may be created by hyperresolution. Indeed, if one refutes the set of clauses $\{a \vee b, \neg a \vee b, a \vee \neg b, \neg a \vee \neg b\}$ using hyperresolution, the corresponding binary resolution tree would not be regular. However, one can insist that the clause tree produced not have any unchosen merge paths on positive literals, nor any tautology path such that the head is a positive literal and is visible from the tail which is the negation of the literal of the head.

**Theorem 4.** *Consider any R/A hyperresolution procedure that is fair, except that it rejects any clause three that contains an unchosen positive merge path or any tautology path whose head is positive. Then the procedure is complete. Moreover, any clause tree that does not contain an unchosen positive merge path nor a tautology path whose head is positive is constructed exactly once by the procedure.*

In effect, half the minimal restriction can be combined with hyperresolution and RA, while retaining completeness and also uniqueness.

## 5 Subsumption with Rank/Activity and Minimality

Non-minimality and activity both are properties that prevent the construction of clause trees that would be removed by subsumption. Subsumption can be an expensive check because it depends on the set of retained clauses, which may be large. Wos[10] refers to this:

> If a strategy could be found whose use prevented a reasoning program from deducing redundant clauses, we would have a solution far preferable to our current one of using subsumption.

Minimality and R/A provide a partial solution, but do not remove every subsumed clause. For instance there may be redundancy in the input clause set, so that the same clause is derived from different input clauses. The question is, can the restrictions of minimality and R/A be used in conjunction with subsumption? The answer is only partially. These different techniques interfere with each other, and lose completeness. In Figure 6, the fair selection strategy is guaranteed by constructing all clause trees of one clause node, of two clause nodes, *et cetera.* Ranks are indicated by numbers in superscript, and an inactive node is denoted by $*$.

| | | |
|---|---|---|
| 1. | $p^1 + \bullet$ | |
| 2. | $a^1 - \bullet - b^2$ | input clause subsumed by 7. |
| 3. | $a^2 + \bullet - p^1$ | input clause subsumed by 6. |
| 4. | $b^2 + \bullet - p^1$ | input clause subsumed by 5. |
| 5. | $b^3 + \bullet - p + \bullet$ | 1. res 4. |
| 6. | $a^3 + \bullet - p + \bullet$ | 1. res 3. |
| 7. | $a^* - \bullet - b + \bullet - p + \bullet$ | 2. res 5. |
| 8. | $b^4 - \bullet - a + \bullet - p + \bullet$ | 2. res 6. |

No more minimal resolutions are possible.

**Fig. 6.** Subsumption interacts with minimality and rank/activity

We can extend any bottom up clause tree procedure to use subsumption fully and maintain completeness, at the cost of losing some of the advantages of minimality and activity. Whenever a clause tree $T$ subsumes a clause tree $T'$, remove both $T$ and $T'$, and replace both with the elementary clause tree of a new input clause $cl(T)$. The ranks of the leaves of this clause tree are assigned as for any input clause. All of these leaves must be deemed active. We call such elementary clause trees *contracted* and we call this subsumption *contracting subsumption.* Figure 7 shows the same example as Figure 6, but using contracting subsumption.

Contracting subsumption retains completeness, but each contracted clause tree it introduces has lost the internal structure that allows non-minimality to be detected, and all leaves in the new clauses are active. Thus the derivations are no longer unique. However, we can avoid some of this redundancy, because not

|  |  |  |  |
|---|---|---|---|
| 1. | $p^1 + \bullet$ | input clause |  |
| 2. | $a^1 - \bullet - b^2$ | input clause | subsumed by 8. |
| 3. | $a^2 + \bullet - p^1$ | input clause | subsumed by 6. |
| 4. | $b^2 + \bullet - p^1$ | input clause | subsumed by 5. |
| 5. | $b^3 + \bullet - p + \bullet$ | 1. res 4. | becomes $b^1 + \bullet$, subsumes 4. |
| 6. | $a^3 + \bullet - p + \bullet$ | 1. res 3. | becomes $a^1 + \bullet$, subsumes 3. |
| 7. | $a^* - \bullet - b + \bullet$ | 2. res 5. | inactive, can be ignored |
| 8. | $b^3 - \bullet - a + \bullet$ | 2. res 6. | becomes $b^1 + \bullet$, subsumes 2. |
| 9. | $\bullet - b + \bullet$ | 5. res 8. | Done. |

**Fig. 7.** Contracting subsumption works with minimality and rank/activity

all subsumptions need a contracting clause. We compute a size for each clause tree. If the subsumed clause is bigger than the subsuming clause, the subsumed clause can be safely rejected, just as the usual form of subsumption. Only if the subsumed clause is the same size or smaller, do we replace it by a contracted clause.

**Definition 6 (Strict Increasing Subsumption).** *A clause tree $T$* subsumes *a clause tree $T^*$ if $cl(T)$ subsumes $cl(T^*)$. It is called* strict increasing subsumption *if $size(T) < size(T^*)$.*

There are various size functions that could be used: the number of clause nodes, the number of edges, the height of the tree, the total size where each clause node is given a weight, *et cetera.* For a given problem and selection strategy, different weight functions would give different proportions of contracting subsumptions.

**Definition 7 (Properties of size functions).** *A size function is* consistent *if $size(T_1) < size(T_2)$ implies that $size(T_1 \text{ res } T) < size(T_2 \text{ res } T)$. We say that a size function is* stable *if, for each clause tree $T$, all admissible derivations of $T$ agree on the size of $T$. A size function is* increasing *if $size(T_1 \text{ res } T_2) > max(size(T_1), size(T_2))$. It is* additive *if $size(T_1 \text{ res } T_2) = size(T_1) + size(T_2)$.*

**Theorem 5 (Completeness with all properties).** *Given a fair minimal clause tree procedure that uses the R/A restriction, uses increasing subsumption and contracting subsumption otherwise, where the size function is increasing, consistent and stable. Moreover suppose that the rank order of literals in the child clause is totally determined by the rank order in the parent clauses. Then this procedure is refutationally complete.*

*Proof.* The proof is a straightforward induction, omitted to save space.

## 6 Discussion

The R/A restriction prevents any derivation from being found twice in the search for a derivation of an empty clause. This is unlike most restrictions of resolution which cut out large portions of the search space, but which allow many different ways to get to any given restriction. We have shown that R/A can be combined with both SOS and hyperresolution, and obtain both types of reduction in the search space. Combining R/A with subsumption is not as clean, but is also possible, gaining all the power of subsumption with some of the power of the rank/activity.

Each time a resolution is to be considered, if an activity check is made and one of the pair of literals is found to be inactive, a resolution (and probably a subsumption check) is saved. On the other hand, R/A may not be very compatible with some of the standard heuristics in automated theorem provers. Because there is only one path that leads to a given refutation, if one of the steps does not score well with some heuristic, that refutation will not be found for a large number of steps. For example, if a procedure always chooses small clauses before larger ones, and if the unique path to a given refutation contains a large clause, then that refutation will not be found until all smaller clauses have been exhausted. Since the smaller clauses may never be exhausted, such a heuristic can lead to an unfair procedure, and completeness can be lost. On the other hand, different heuristics, which could take the R/A restriction into account, can be used instead. Such a heuristic could prefer smaller clause trees, at least to some extent. Implementations are required to test out various heuristics.

## References

1. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York and London, 1973.
2. Hans de Nivelle, 1998. Personal communication.
3. J. D. Horton and B. Spencer. Clause trees: a tool for understanding and implementing resolution in automated reasoning. *Artificial Intelligence*, 92:25–89, 1997.
4. J. D. Horton and B. Spencer. Rank/activity: a canonical form for binary resolution. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – Cade-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 412–426. Springer-Verlag, Berlin, July 1998.
5. L. Wos, D. Carson and G. Robinson. Efficiency, completeness and the set of support strategy in theorem proving. *J. ACM*, 12:536–541, 1965.
6. W. W. McCune. Otter 3.0 users guide. Technical Report ANL-94/6, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL, 1994.
7. J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
8. Bruce Spencer and J.D. Horton. Extending the regular restriction of resolution to non-linear subdeductions. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 478–483. AAAI Press/MIT Press, 1997.
9. G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics*, Seminars in Mathematics: Matematicheskii Institute, pages 115–125. Consultants Bureau, 1969.
10. Larry Wos. *Automated Reasoning : 33 Basic Research Problems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.