Advanced Indexing Operations on Substitution Trees

Peter Graf*, Christoph Meyer

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
email: peter.graf@sap-ag.de, meyer@mpi-sb.mpg.de

Abstract. Indexing techniques support the retrieval and maintenance of large sets of terms. There is also an indexing method called substitution tree indexing that efficiently handles sets of substitutions. We present three advanced indexing operations for substitution trees: The multi-merge for the simultaneous unification of sets of substitutions, the subsumption operation on two sets of substitutions, and the selection of 'lightest' substitutions of a set of substitutions. The indexing operations can be combined to obtain powerful reasoning tools for theorem provers.

1 Introduction

Theorem provers that implement synthetic calculi like resolution [10, 2] face the problem of *program degradation*: The theorem prover's rate of drawing conclusions falls off sharply with time due to an increasing amount of retained information [11].

Term indexing particularly influences a system's performance by providing rapid access to first-order predicate calculus terms with specific properties. Typical queries to a logical database [1] in context with theorem proving are: Given a database $\mathcal{I}$ containing terms (literals) and a single query term t, find all terms in $\mathcal{I}$ that are unifiable with, instances of, or more general than t. Thus standard applications of term indexing are the search of resolution partners for a given term (literal) or the retrieval of literals in clauses for both forward and backward subsumption.

The standard approaches in term indexing work for a single query term for which partners in an indexed set of terms are searched. Our advanced indexing operations are able to handle sets of query terms at a time. In this work we shall demonstrate how advanced indexing operations can support the search of simultaneous unifiers in sets of substitutions as it is necessary for unit resulting resolution [5] or hyperresolution [9], for example. Moreover, a subsumption procedure on two indexes as well as a selection mechanism for 'lightest' entries will be presented.

* This work was supported by the German Science Foundation (DFG). Peter Graf's current address is SAP AG, 69185 Walldorf, Germany.

The advanced indexing operations are based on a specific indexing technique called *substitution tree indexing* [3]. This method does not only provide an efficient representation of terms but also of substitutions.

As an example, Fig. 1 illustrates the efficient implementation of unit resulting resolution using our advanced indexing operations. We attach substitution trees to each literal of the nuclei in the initial problem set. The trees represent sets Σ_i of substitutions. Each substitution of a set Σ_i instantiates the according literal L_i. Here it is of advantage if the indexing technique employed is able to index substitutions in a convenient manner.

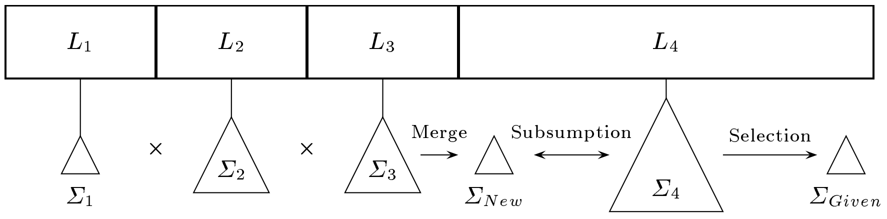

Fig. 1. UR-Resolution on a nucleus $\{L_1, L_2, L_3, L_4\}$

The set Σ_1 contains the unifiers of the according literal L_1 with electrons which have not been combined yet with L_1. The sets Σ_2 and Σ_3 contain unifiers which have been considered in previous steps. The result of the simultaneous unification of the three sets is the set Σ_{New} of substitutions containing the common instances σ representing new electrons $L_4\sigma$. The simultaneous unification of an arbitrary number of substitution trees can be achieved by the so-called multi-merge operation presented in this paper.

The subsumption test of the set Σ_{New} with previously generated electrons contained in the substitution set Σ_4 is an application for our subsumption operation on two indexes. Usually, not all of the produced electrons can be taken into account for subsequent ur-resolution steps. Therefore, we provide an efficient operation for the selection of 'lightest' substitutions resulting in a substitution tree Σ_{Given}.

This approach has been implemented in a distributed theorem prover called PURR [7] (Parallel Unit Resulting Resolution).

In the second and third section we present some preliminaries and a classification of indexing techniques. Section 4 contains an introduction to substitution tree indexing. The advanced indexing operations discussed in Sect. 5 are simultaneous unification of substitutions, subsumption of substitutions, and selection of 'lightest' substitutions. In Sect. 6 the results of several experiments will be presented.

2 Preliminaries

The standard notions for first order logic are used. $\mathbf{F}_n$ is the set of n-ary *function symbols*, $\mathbf{V}$ denotes the set of *variable* symbols and $\mathbf{V}^* \subset \mathbf{V}$ is the set of *indicator variables*. The variables that occur in a term or a set of terms are denoted by $\mathsf{VAR}(t)$. In our examples the symbols $u, v, w, x, y, z \in \mathbf{V}$ and $*_i \in \mathbf{V}^*$ are used for variables. The symbols f, g, h denote function symbols and a, b, c denote constants, i.e. $a, b, c \in \mathbf{F}_0$.

The set $\mathsf{DOM}(\sigma) := \{x \in \mathbf{V} \mid x\sigma \neq x\}$ is called *domain* of the substitution σ, the set $\mathsf{COD}(\sigma) := \{x\sigma \mid x \in \mathsf{DOM}(\sigma)\}$ the *codomain* of σ, and $\mathsf{IM}(\sigma) := \mathsf{VAR}(\mathsf{COD}(\sigma))$ is the set of variables *introduced* by σ. The *composition* $\sigma\tau$ of substitutions $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ and $\tau = \{y_1 \mapsto t_1, \ldots, y_m \mapsto t_m\}$ is defined as $x(\sigma\tau) := (x\sigma)\tau$ for all x. The *join* of the substitutions σ and τ is defined as $\sigma \bullet \tau := \{x_1 \mapsto s_1\tau, \ldots, x_n \mapsto s_n\tau\} \cup \{y_i \mapsto t_i \mid y_i \in \mathsf{DOM}(\tau) \backslash \mathsf{IM}(\sigma)\}$. For $\sigma = \{z \mapsto g(x)\}$ and $\tau = \{x \mapsto a, y \mapsto c\}$ we have $\sigma\tau = \{z \mapsto g(a), x \mapsto a, y \mapsto c\}$ and $\sigma \bullet \tau = \{z \mapsto g(a), y \mapsto c\}$.

3 Classifications of Term Indexing Techniques

Relations on Terms and Substitutions. The main purpose of indexing techniques in theorem provers is to achieve efficient access to first-order terms with specific properties. To this end, a set of terms $\mathcal{I}$ is inserted into an indexing data structure. A retrieval in $\mathcal{I}$ is started for a set $\mathcal{Q}$ of *query terms*. The aim of the retrieval is to find tuples (s, t) with $s \in \mathcal{I}$ and $t \in \mathcal{Q}$ in such a way that a special relation R holds for s and t. Most automated reasoning systems can profit from a retrieval based on the following relations: s and t are *unifiable*, t is an *instance* of s, and s is a *generalization* of t. If we are interested in retrieving indexed *substitutions* instead of indexed terms, a generalized relation $R(\sigma, \tau)$ is needed.

Retrieval of Type 1:1, n:1, and n:m. A retrieval is of type 1:1 if both sets $\mathcal{I}$ and $\mathcal{Q}$ have cardinality 1. Since both sets $\mathcal{Q}$ and $\mathcal{I}$ solely consist of one single term or substitution, the retrieval corresponds to simply testing if $R(s, t)$ holds.

Retrieval of type n:1 is determined by a single query term t, which is used to find entries $s \in \mathcal{I}$. The set $\mathcal{I}$ of n indexed terms is represented by an indexing data structure. The result of a retrieval is a subset of $\mathcal{I}$. Note that a very inefficient retrieval of type n:1 could be performed by testing each entry of the index in a 1:1 type retrieval because such an approach would have to consider all indexed terms explicitly.

Retrieval of type n:m includes all cases in which more than a single query term is involved. Exploiting n:m indexing, the query set typically is also represented by an index. Hence, we have to deal with two indexes; one of them represents the indexed and the other one represents the query set. The result of such a retrieval is a subset of the direct product of the term sets involved.

Maintenance of Type n:1 and n:m. In addition to the retrieval operations we also have to provide functions that *insert* entries into and *delete* entries from the indexing structure. Insertion and deletion can also be classified according to the cardinalities of the involved sets.

Maintenance of type n:1 includes all operations that modify an index by a single term. Beside the classical insertion and deletion operations of a single term, the deletion of all instances of a term, for example, also corresponds to an n:1 maintenance operation.

Maintenance of type n:m corresponds to index manipulation operations that fit into the concept of n:m indexing. For example, the union of two indexes results in a new index that contains all terms of the two sets involved. An additional n:m maintenance task is to delete all instances of $\mathcal{Q}$ that occur in $\mathcal{I}$ from $\mathcal{I}$. Such an operation is used for subsumption in the case of unit clauses, for example.

4 Substitution Tree Indexing

Substitution tree indexing is an indexing technique that has been developed from discrimination tree indexing [6] and abstraction tree indexing [8]. A substitution tree (ST) can represent any set of idempotent substitutions. In the simplest case all these substitutions have identical domains and consist of a single assignment, which implies that the substitution tree can be used as a term index as well. Fig. 2 shows an index for the substitutions $\{x \mapsto f(a,b), y \mapsto c\}$ and $\{x \mapsto f(u,b), z \mapsto d\}$. As the name indicates, the labels of substitution tree nodes are substitutions. Each branch in the tree represents a binding chain for variables. Consequently, the substitutions of a branch from the root down to a particular node can be composed and yield an instance of the root node's substitution. Consider the substitution $\tau = \{x \mapsto f(a,b), y \mapsto c\}$, which is represented by the chain of substitutions $\tau_0 = \{x \mapsto f(x_1,b)\}$, $\tau_1 = \{x_1 \mapsto a, y \mapsto c\}$. The original substitution τ can be reconstructed by simply applying the substitution $\tau_0\tau_1$ to the domain of τ. The result of this application is $\tau = \{x \mapsto x(\tau_0\tau_1), y \mapsto y(\tau_0\tau_1)\} = \{x \mapsto f(x_1,b)\tau_1, y \mapsto y\tau_1\} = \{x \mapsto f(a,b), y \mapsto c\}$.

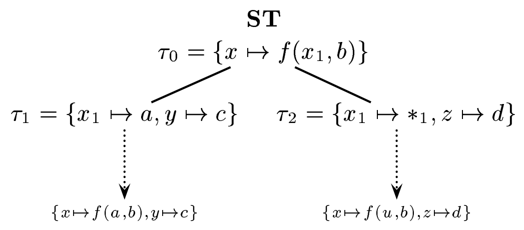
ST
$\tau_0 = \{x \mapsto f(x_1,b)\}$
$\tau_1 = \{x_1 \mapsto a, y \mapsto c\}$
$\tau_2 = \{x_1 \mapsto *_1, z \mapsto d\}$
$\{x \mapsto f(a,b), y \mapsto c\}$
$\{x \mapsto f(u,b), z \mapsto d\}$

Fig. 2. Representation of substitutions

Substitutions are *normalized* [3] before being inserted into the index. Normalization renames all variables in the codomain to so-called *indicator variables*, which are denoted by $*_i$. The substitutions represented by the index in Fig. 2 therefore are $\{x \mapsto f(a,b), y \mapsto c\}$ and $\{x \mapsto f(*_1,b), z \mapsto d\}$. The renaming is done for two main reasons: There is more structure sharing in the index if the substitutions are normalized and, when searching for instances in the index, indicator variables must not be instantiated and therefore need to be marked specially.

Retrieval in substitution trees is based on a backtracking algorithm in addition to an ordinary representation of substitutions as lists of variable-term pairs.

The retrieval also needs a backtrackable variable binding mechanism, similar to the one used in PROLOG.

Definition 1 (Substitution Tree). A *substitution tree* is a tuple (τ, Σ) where τ is a substitution and Σ is an ordered set of substitution trees. The following conditions hold:

- A substitution tree is either a *leaf* $(\tau, \emptyset)$ or $|\Sigma| \geq 2$.
- For every path $(\tau_1, \Sigma_1), \ldots, (\tau_n, \emptyset)$ starting from the root of a substitution tree we have $\mathsf{IM}(\tau_1 \bullet \ldots \bullet \tau_n) \subset \mathbf{V}^*$.
- For every path $(\tau_1, \Sigma_1), \ldots, (\tau_i, \Sigma_i)$ from the root to any node of a tree we have $\mathsf{DOM}(\tau_i) \cap \bigcup_{1 \leq j < i} \mathsf{DOM}(\tau_j) = \emptyset$.

If $(\tau_1, \Sigma_1), \ldots, (\tau_n, \Sigma_n)$ is a path from the root of the tree to node (τ_n, Σ_n) and x occurs in the codomains of the τ_i but not in the domains of the τ_i, then the variable x is called *open* at node (τ_n, Σ_n). Variables that are not open at a node N are called *closed* at N. The second condition in Def. 1 implies that all non-indicator variables are closed at leaf nodes of substitution trees. The *empty* tree is denoted by ε.

5 Advanced Indexing Operations

We call *n*:*m* retrieval and maintenance tasks "advanced indexing operations". In this chapter several advanced indexing operations are considered: The *multi-merge* operation computes simultaneous unifiers of substitutions which are stored in several substitution trees. The result of such a multi-merge is a substitution tree containing the common instances of the unified substitutions. The subsumption operation deletes in one substitution tree all instances of substitutions that occur in another substitution tree. The selection operation searches a substitution tree for entries with lowest "weight" and adds these entries to another substitution tree.

5.1 Multi-Merge

Standard Merge. The merge [8] operation computes the compatible substitutions stored in two different trees. Substitutions are compatible if the codomains of identical variables in the two substitutions are simultaneously unifiable.

Suppose we want to merge three substitution trees M, N, and O. Using the ordinary merge operation for two trees, we first merge M and N. The resulting tree which contains the most general common instances of M and N is finally merged with O. However, the merge does not necessarily have to be performed on just two trees in a single merge operation. Instead of performing two merges and creating an intermediate result, we use a new backtracking algorithm that traverses the three trees in parallel. In this way, we avoid the creation of intermediate results and thus save a large amount of memory.

Multi-Merge. The multi-merge operation [4] takes an arbitrary number of substitution trees and traverses the trees in parallel. If a combination of leaf nodes is reached, the resulting common instance of the substitutions represented by these leaves can be stored in a new substitution tree. Furthermore, subsumption might be performed thus reducing the amount of substitutions to be maintained.

Consider the algorithm `multimerge` in Fig. 3 which employs n:1 insertion and subsumption operations. The algorithm has four parameters: A substitution tree RES, two ordered sets $CURRENT$ and $NEXT$ of substitution trees, and a stack STK of bindings. The common instances resulting from the simultaneous unification are inserted into the substitution tree RES which does not have to be empty at the beginning. The tree may contain previously obtained results which are then considered in the subsumption phase of the multi-merge operation. Initially, the ordered set $CURRENT$ contains the substitution trees to be merged whereas the ordered set $NEXT$ is empty. We assume that the substitutions of the root nodes have been successfully unified before `multimerge` is called. In this way we avoid unnecessary recursive calls in the algorithm. The variable bindings of the unification are pushed on the stack STK.

```
algorithm multimerge(tree $RES$, set $CURRENT$, set $NEXT$, stack $STK$)
 begin
    ⟨ Let $CURRENT = \{N_i, \ldots, N_m\}$ be an ordered set of trees ⟩
    ⟨ Let $NEXT = \{N_1, \ldots, N_{i-1}\}$ be an ordered set of trees ⟩
    if $\forall N \in CURRENT \cup NEXT$ : is_leaf($N$) then
        ⟨ Simultaneous Unifier ⟩
        if $\neg$genexist($RES, STK$) do
            delete_instances($RES, STK$)
            $RES$ = insert($RES, STK$)
    else
        if $CURRENT = \emptyset$ then
            $RES$ = multimerge($RES, NEXT, CURRENT, STK$)
        elsif is_leaf($N_i$) then
            $RES$ = multimerge($RES, CURRENT \setminus N_i, NEXT \cup \{N_i\}, STK$)
        else
            ⟨ Let $(\tau, \Sigma) = N_i$ be the root of $N_i$ ⟩
            forall $N' \in \Sigma$ do
                if unify($N', STK, BINDINGS$) then
                    $RES$ = multimerge($RES, CURRENT \setminus N_i,$
                                            $NEXT \cup \{N'\}, STK$)
                backtrack($STK, BINDINGS$)
    return $RES$
 end
```

Fig. 3. Algorithm for `multimerge`

The function unify($N, STK, BINDINGS$) implements the test for unifiability by checking for each assignment $x_i \mapsto t_i$ of N's substitution $\tau = \{\ldots, x_i \mapsto t_i, \ldots\}$ whether x_i is unifiable with t_i. The bindings of variables in the unifier are pushed on the stack STK and are counted in $BINDINGS$. This unification considers variable bindings in the terms to be unified. Additionally, the function backtrack($STK, BINDINGS$) resets the stack STK by popping $BINDINGS$ bindings from it. After a successful unification at leaf nodes the function genexist performs n:1 forward subsumption using the established bindings. If no generalization of the found common instance σ exists in RES, the function delete_instances removes all instances of σ from RES by a n:1 backward subsumption. Finally, the function insert normalizes and inserts σ into the substitution tree RES. Note that all functions work with bindings instead of really instantiated substitutions. In this way we delay (and often avoid) the allocation of memory as long as possible.

The main idea of the algorithm `multimerge` is to traverse the trees in parallel. All combinations of subnodes of the $CURRENT$ set of inner nodes have to be considered. The subnodes which pass the test for unifiability are moved to the $NEXT$ set of nodes (s. line 19). If $CURRENT$ is empty we simply exchange $CURRENT$ with the $NEXT$ level (s. line 12). $CURRENT$ leaf nodes are also moved to the $NEXT$ level in order to uphold the original order of trees (s. line 14). Each combination of leaf nodes represents a simultaneous unifier which corresponds to the established bindings on the stack STK (s. line 5).

A sequence of stacks resulting from the simultaneous unification of substitutions stored in three substitution trees is depicted in Fig. 4. Originally, the stack is empty. Before we start the multi-merge algorithm, the substitutions of the root nodes have to be unified, resulting in the bindings pushed on the stack (compare stack "Init"). The sequence $\textcircled{A}\textcircled{U}\textcircled{X}$ denotes the tree nodes which have been considered in this step. The recursive algorithm is started on the subnodes of the root nodes. In case it succeeds in testing the current substitution for unifiability, the modified stack is marked with "Success". If a combination of leaf nodes has been found, "**Success**" is written boldface. The first common instance is $\{u \mapsto f(d, g(d))\}$ which is backward subsumed by the second common instance $\{u \mapsto f(v, g(v))\}$. The last found substitution $\{u \mapsto f(b, g(b))\}$ is forward subsumed by the second. Therefore, the result of the multi-merge is a substitution tree only containing $\{u \mapsto f(v, g(v))\}$.

5.2 Subsumption

In resolution-based theorem provers subsumption is a powerful technique for pruning the search space. The *forward* subsumption test checks for a given clause C if the set of kept clauses contains a generalization of C. In this case, clause C may be discarded. Otherwise, before we insert clause C into the set of kept clauses, all instances of C in the set can be removed. We refer to this operation as the *backward* subsumption.

Subsumption can be supported by indexing techniques. In general, forward subsumption corresponds to an n:1 *retrieval task* and backward subsumption to an n:1 *maintenance task*. If we deal with unit clauses, for example when performing ur-resolution, both subsumption tasks can be performed by a complex n:m deletion operation on substitution trees that represent these unit clauses.

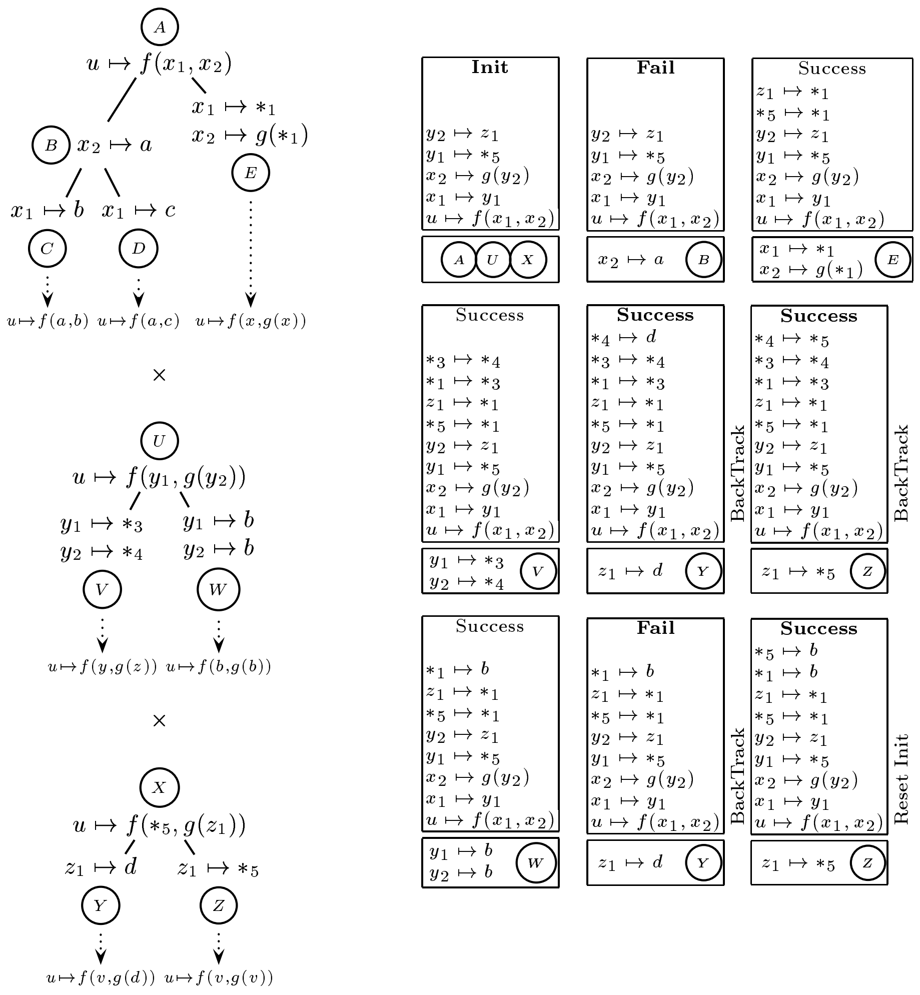
Fig. 4. Multi-Merge with three substitution trees

Suppose the set of kept unit clauses is represented by an index N. Furthermore, we have an index M containing a set of new unit clauses which are now tested for subsumption. Backward subsumption corresponds to the deletion of all instances of M in N. To this end, we traverse M and N in parallel. During the traversal we map variables occurring in M to subterms stored in N. This mapping is exactly the same as just looking for instances in a n:1 retrieval. Whenever we reach a leaf node in the index N we may delete it. Note that the deletion in the index N can cause the whole tree to be removed.

We obtain forward subsumption by simply changing the roles of M and N. After subsumption the unit clauses contained in M are usually inserted into index N. Therefore, Graf developed a so-called *union* [4] of indexes which inserts a whole index into another.

Consider the algorithm `subsume` depicted in Fig. 5. Subsumption has to consider three major situations occurring during the traversal of the trees. First, in tree M we may find a leaf node. In this situation we have to check if tree N contains a generalization of the current bindings in the corresponding subtree. If this is the case, the leaf node in M is deleted (s. line 6). Second, tree M is not a leaf node, but the corresponding node in N is. Here we simply call a deletion routine that deletes all instances of the current bindings in M (s. line 8). Third, if two inner nodes are considered, we proceed by considering all possible combinations of subnodes until tree M has been completely deleted or no more combinations are available. Note that in line 17 node M has to be "repaired" if all subtrees of M but one have been deleted[1].

```
 1  algorithm subsume(tree $M$, tree $N$, stack $STK$)
 2   begin
 3      $\langle$Assume match_reverse($M$) and match($N$) hold$\rangle$
 4      if is_leaf($M$) then
 5          if genexist($N, STK$) then
 6              $M = \varepsilon$
 7      elsif is_leaf($N$) then
 8          $M =$ delete_instances($M, STK$)
 9      else
10          forall subtrees $M'$ of $M$ do
11              if match_reverse($M', STK, BINDINGS^M$) then
12                  forall subtrees $N'$ of $N$ do
13                      if match($N', STK, BINDINGS^N$) then
14                          $\Sigma = \Sigma \cup$ subsume($M', N', STK$)
15                      backtrack($STK, BINDINGS^N$)
16              backtrack($STK, BINDINGS^M$)
17          $M =$ repair($M, \Sigma$)
18      return $M$
19   end
```

Fig. 5. Algorithm for `subsume`

For example, suppose we have to deal with the two trees M and N depicted in Fig. 6. We compute the tree M' resulting from $\mathtt{subsume}(M, N, STK)$. In tree M we maintain the substitutions $\{u \mapsto f(a,b)\}$, $\{u \mapsto f(x,c)\}$, and $\{u \mapsto f(d,c)\}$. Tree N contains the substitutions $\{u \mapsto f(a,c)\}$, $\{u \mapsto f(a,y)\}$, and $\{u \mapsto f(z,c)\}$. Obviously, the substitution $\{u \mapsto f(a,y)\}$ stored in N subsumes the substitution $\{u \mapsto f(a,b)\}$ stored in M. Moreover, $\{u \mapsto f(z,c)\}$ subsumes the two substitutions $\{u \mapsto f(x,c)\}$ and $\{u \mapsto f(d,c)\}$. Hence, tree M' resulting from subsumption should be empty.

The function $\mathsf{match}(N, STK, BINDINGS)$ checks if N's substitution is a generalization of the current bindings. New bindings are established on the stack.

[1] According to definition 1 of substitution trees an inner node has at least two sons.

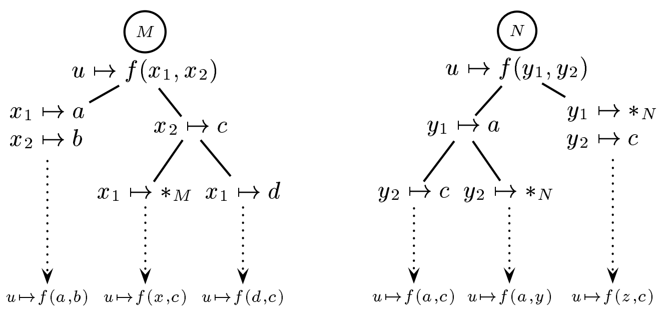

Fig. 6. Subsumption as an n:m indexing task

The function match_reverse$(M, STK, BINDINGS)$ is accordingly defined to test whether M's substitution is an instance of the bindings. We start at the root nodes of tree M and tree N and assume that match_reverse$(M, STK, BINDINGS)$ and match$(N, STK, BINDINGS)$ have been successfully called before we enter the subsumption operation. The initialization establishes the binding $\{u \mapsto f(x_1, x_2)\}$ with the root of M and the bindings $\{y_1 \mapsto x_1, y_2 \mapsto x_2\}$ with N. Using the algorithm `subsume` we recursively traverse the subtrees. First, we consider the left subtree in M where the test function match_reverse yields the bindings $\{x_1 \mapsto a, x_2 \mapsto b\}$. Keeping the current bindings, the left subtree of N is traversed, searching for leaf nodes that correspond to substitutions more general than $\{u \mapsto f(a, b)\}$. Such a substitution is found in $\{u \mapsto f(a, y)\}$ and the leaf node representing $\{u \mapsto f(a, b)\}$ is deleted. Now, the right subtree of M is considered establishing the binding $\{x_2 \mapsto c\}$. Since the left subtree of N does not yield any deletions (because the test function match_reverse must not bind the variable $*_M$), we immediately consider the right subtree of N, which is a leaf marked with $\{y_1 \mapsto *_N, y_2 \mapsto c\}$. Applying function match on this node yields the binding $\{x_1 \mapsto *_N\}$. According to the definition of `subsume` we delete all instances in the right subtree of M. Finally, both subtrees of M have been deleted and repairing the resulting tree leads to ε, i.e. tree M has been deleted completely.

5.3 Selection

A heuristic that is used in many resolution-based theorem provers selects the smallest clauses in the set of kept clauses for the application of inference rules. In general, we call a clause A *lighter* than a clause B if A has less *weight* than B. The weight of a clause is determined by a *weighting function*.

For example, if the weighting function counts the number of symbols in a clause then the clause $C = \{P(f(x, y))\}$ has weight 4 and the clause $D = \{P(g(a))\}$ has weight 3. Following the heuristic clause D would be selected first. The selection of lightest substitutions in a substitution tree is of specific interest if these substitutions represent unit clauses for ur-resolution. Following our heuristic only the lightest clauses serve as partners in inference steps. Thus the lightest substitutions in the tree should be selected only once.

The selection is a $n{:}m$ maintenance operation on two indexes M and N. The set of lightest substitutions occurring in N is added to M. The selected substitutions are not removed from N, but marked as selected preventing multiple selection. The marked substitutions in N are still considered for conventional retrieval. To provide this functionality the nodes of a substitution tree are modified such that:

1. A leaf node $(\tau, \emptyset, w, s)$ refers to entries with the corresponding weight w. The *state* s is true if all entries have been selected.
2. The weight w of an inner node (τ, Σ, w, s) is the minimum of the weights of the subtrees Σ with unselected substitutions. The state s is true if all entries of the subtrees Σ have been selected. In this case the weight w is arbitrary.

A sequence of substitution trees is depicted in Fig. 7. In tree $\textcircled{1}$ no entries have been selected yet. The lightest entries have weight 2. Note that a tree contains unselected entries if the state s of the root node is marked false. Then the weight w of the root node corresponds to the weight of the lightest unselected entry in the tree.

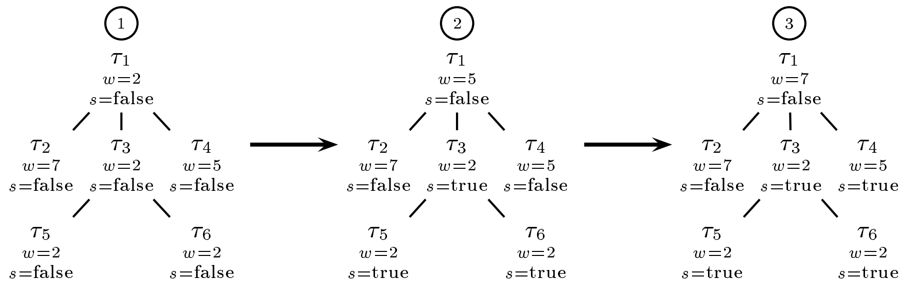

Fig. 7. A sequence of substitution trees resulting from selection

In tree $\textcircled{2}$ the two lightest entries have been selected. The subtree of τ_3 is marked as selected. The new minimal weight 5 has been propagated to the root. Finally, the entry with weight 5 has been selected in tree $\textcircled{3}$.

The selection considers a substitution tree N as being separated into partitions of different weights. Each partition contains substitutions with identical weight. The lightest partition can be selected within a single retrieval operation. Note that this operation also modifies tree N to provide consistency of N (In particular, a new lightest weight w and the state s are propagated to the root). If the lightest partition is completely retrieved, the weight w in the root of N corresponds to the weight of the new lightest partition. Thus a single selection operation retrieves at most the substitutions of the lightest partition.

The function `selection` depicted in Fig. 8 implements a selection of n substitutions in a substitution tree N and stores the retrieved substitutions in tree M. The auxiliary function `partition` repeatedly retrieves the lightest partition in N (s. line 4). Thus the number n of demanded substitutions does not depend on the size of the lightest partition in N.

```
1   **algorithm** selection(**tree** $M$,**tree** $N$,**nat** $n$,**stack** $STK$)
2    **begin**
3        **while** $n > 0 \wedge N$ *is unselected* **do**
4            $(M, N, n) =$ partition$(M, N, n, STK)$
5        **return** $(M, N, n)$
6    **end**
```

Fig. 8. Algorithm for `selection`

Consider the algorithm for `partition` in Fig. 9. We assume that `partition` is called only with substitution trees containing unselected substitutions. Furthermore, the current tree node N in `partition` always contains substitutions with the lightest weight w, i.e. node N is marked with weight w. Therefore, a parameter for the lightest weight in tree N is not needed in the algorithm.

```
1   **algorithm** partition(**tree** $M$,**tree** $N$,**nat** $n$,**stack** $STK$)
2    **begin**
3        ⟨ *Assume that N has unselected entries* ⟩
4        bind$(N, STK, \mathrm{BINDINGS})$
5        **if** is_leaf$(N)$ **then**
6            $\langle N = (\tau, \emptyset, w, \mathsf{false})\rangle$
7            $M =$ insert$(M, STK)$
8            backtrack$(STK, BINDINGS)$
9            **return** $(M, (\tau, \emptyset, w, \mathsf{true}), n - 1)$
10       **else**
11           $\langle N = (\tau, \Sigma, w, \mathsf{false})\rangle$
12           **forall** subtrees $N'$ of $N$ **do**
13               **if** $n > 0 \wedge N'$ *is unselected with weight* $w$ **then**
14                   $(M, N', n) =$ partition$(M, N', n, STK)$
15               $\Sigma' = \Sigma' \cup N'$
16           backtrack$(STK, BINDINGS)$
17           **return** $(M, (\tau, \Sigma', \mathsf{lightest\_weight}(\Sigma'), \mathsf{all\_selected}(\Sigma')), n)$
18   **end**
```

Fig. 9. Algorithm for `partition`

In order to reconstruct the selected substitutions in tree N, the function bind establishes N's substitution on the stack (s. line 4). If node N is a leaf, then insert adds the corresponding substitution, which is represented by the established bindings, to tree M (s. line 7). Note that leaf N will be marked as selected (s. line 9). If node N is an inner node, then the subtrees of N containing unselected entries with the lightest weight w are recursively searched (s. line 14). As the algorithm propagates a new lightest weight and the state of selection from the leaves to the root, the set Σ' contains the updated subtrees of N (s. line 15). The new weight of node N is the lightest weight of the subtrees in Σ' with unselected entries computed by the function lightest_weight. The function all_selected is true if all entries in all subtrees in Σ' have been selected (s. line 17).

Again, consider the example depicted in Fig. 7. It shows the selection of three entries. The substitution tree contains three partitions. The lightest partition with weight 2 contains two entries. The first call of `partition` results in the second tree. In the rightmost tree the second partition with weight 5 has been selected by the second call of `partition`.

6 Experiments

Subsumption. The experiments in the left table of Fig. 10 were run on a Sun SPARCstation SLC computer with 16 MBytes of RAM. The sets EC (500), CL (1000), and BO (6000) contain substitutions with a single domain variable u like $\{u \mapsto f(a,x)\}$, for example. The number in brackets refers to the number of substitutions in the set. The sets were taken from real OTTER applications [6]. The indexing problem EC+− for n:1 forward subsumption refers to storing the set EC+ in an index and checking the existance of generalizations in the index for each member of the query set EC−. The n:m operation also maintains an index for EC− and deletes all instances of generalizations stored in EC+ from EC−. For the randomly created sets AVG, WIDE, GND, LIN, and DEEP the index set and the query set are identical. Each set contains 5000 substitutions.

Task	Subsumption				Merge	
	Forward		Backward			
	n:1	n:m	n:1	n:m	n:1	n:m
EC++	**0.3**	0.4	1.1	**0.4**	16.5	**15.3**
EC+−	0.5	**0.1**	1.4	**0.1**	27.8	**7.2**
EC−+	**0**	0.1	0.4	**0.1**	**7.3**	7.9
EC−−	0.7	**0.5**	2.9	**0.5**	61.3	**29.1**
CL++	**0.9**	1.0	2.5	**1.0**	10.2	**6.5**
CL+−	**0.4**	3.6	0	0	**7.3**	12.8
CL−+	0	0	9.6	**3.7**	23.4	**11.3**
CL−−	**1.6**	1.8	4.5	**1.8**	9.2	**5.9**
BO++	**2.6**	3.4	6.2	**3.4**	15.4	**13.5**
BO+−	**3.8**	6.1	3.8	**3.5**	9.0	**7.3**
BO−+	**1.0**	3.5	**3.1**	6.2	**4.4**	8.9
BO−−	1.9	**1.5**	4.4	**1.5**	5.7	**2.8**
AVG	**0.7**	1.4	**1.4**	**1.4**	27.4	**22.6**
WIDE	**5.2**	32.5	**20.1**	32.7	**52.7**	172.7
GND	**1.0**	**1.0**	2.4	**1.0**	3.2	**2.3**
LIN	**0.4**	0.7	1.2	**0.7**	23.7	**18.4**
DEEP	**0.5**	1.1	1.2	**1.1**	56.0	**41.6**

Task		Multi-Merge					
		Seconds			MBytes		
		n:1	n:m	multi	n:1	n:m	multi
AVG	1-2	5.7	**5.6**	17.0	1.3	1.3	0
	2-3	12.3	**11.1**	28.5	2.1	2.1	0
	2-4	80.0	105.0	**24.1**	74.4	76.0	0
	3-4	83.0	87.1	**23.2**	77.8	87.4	0
WIDE	1-2	**0.4**	**0.4**	50.2	0.1	0.1	0
	2-3	**3.6**	7.6	209.9	0	0	0
	2-4	261.0	fail	**258.5**	362.5	fail	0
	3-4	283.6	fail	**129.2**	362.6	fail	0
GND	1-2	**0.7**	0.8	12.5	0.2	0.2	0
	2-3	**1.3**	1.8	9.7	0.1	0.1	0
	2-4	67.8	88.0	**11.5**	70.2	75.0	0
	3-4	69.3	73.5	**13.9**	71.0	77.0	0
LIN	1-2	**8.9**	**8.9**	17.6	1.7	1.9	0
	2-3	**8.1**	8.5	28.2	1.3	1.4	0
	2-4	81.0	104.9	**28.7**	67.6	74.8	0
	3-4	84.8	90.0	**18.2**	73.0	82.0	0
DEEP	1-2	18.0	**16.3**	31.2	5.1	4.8	0
	2-3	8.5	**8.0**	51.2	1.6	1.6	0
	2-4	168.1	252.5	**70.6**	195.8	230.3	0
	3-4	209.2	227.2	**66.4**	204.8	246.4	0

Fig. 10. Experiments with subsumption, merge, and multi-merge

Consider the retrieval times for forward subsumption. As the n:m operation corresponds to a maintenance task which also deletes the found entries from the index representing the query set, the n:1 retrieval task slightly performs better. The n:1 and n:m operations for backward subsumption are both maintenance tasks since the found instances are deleted in both cases. In all but three experiments the n:m operation shows better performance.

Binary Merge. Consider the merge column of the left table in Fig. 10. Here the problem EC+− refers to storing the substitutions of EC+ in an index and to find compatible substitutions for each of the substitutions in EC−. In the case of an n:1 retrieval, the tree containing EC+ is traversed for each entry of EC−. The n:m operation also stores the set of query substitutions EC− in a substitution tree and performs a merge operation on the two substitution trees. In most of the experiments the merge was faster than the standard n:1 retrieval operation.

Multi-Merge. The experiments with the multi-merge were run on a Sun SPARCstation 10 computer with 512 MBytes of RAM. We report experiments with four substitution trees each of which contains 60 randomly created substitutions. The n:1 and n:m operations compute compatible substitutions as described above. The tree resulting from the merge of the first two trees is merged with the third tree and so on. The "multi" operation corresponds to a real multi-merge operation.

The notation "1-2" in the leftmost column means that only the first and the second tree contain substitutions with common domains. In other words, the constellation "1-2" is likely to have less unifiers of the first and the second tree than the order "3-4", for example. Thus the order of trees determines the size of the intermediate results. Note that we could add assignments of an unused domain variable to all substitutions making an optimized preordering of trees difficult.

We observe that the multi-merge operation is the fastest technique on problems with large intermediate results. Another advantage of the multi-merge is that it requires no memory for intermediate results. If the number of unifiers of the first trees is small, the n:1 and n:m operations perform better. In all cases the n:m operation is not faster than the n:1 operation. Note that the performance of the multi-merge seems to be more or less independent from the order of the trees.

Applications. The multi-merge operation supports all types of inferences involving the simultaneous unification of at least two sets of substitutions. Important examples are ur-resolution and hyperresolution. The subsumption test on unit clauses, for example in ur-resolution, is efficiently implemented by our subsumption operation. The selection operation supports weighting heuristics by providing fast access on the lightest entries of a substitution tree.

The presented indexing operations have been implemented in a distributed theorem prover called PURR [7] ($\underline{\text{P}}$arallel $\underline{\text{U}}$nit $\underline{\text{R}}$esulting $\underline{\text{R}}$esolution). All operations in the system are carried out in the framework of indexing. Indexes of substitutions are the fundamental data structure of the prover. The notion of clauses and literals is not required. The system even communicates with indexes.

7 Conclusion

Three indexing operations for substitution trees have been presented. The multi-merge operation supports the simultaneous unification of sets of substitutions. We also included the creation and subsumption of the common instances into the multi-merge, thus making this operation a flexible tool for working with an arbitrary number of indexes. Using the multi-merge the simultaneous unification of substitutions can be achieved either by a repeated binary-merge or by a single multi-merge operation. Indexes of substitutions can efficiently be tested and maintained by the subsumption operation. The selection of lightest entries addresses the need for fast access on the 'best' candidates in an index. As the indexing operations does not only work with but also result in new or modified indexes, these methods can directly be combined to obtain powerful reasoning tools. The algorithms have been implemented and tested in isolation on large sets of substitutions as well as components in a parallel ur-resolution theorem prover.

Acknowledgments. We thank Hans Jürgen Ohlbach and the referees for the comments on earlier versions of this paper.

References

1. R. Butler and R. Overbeek. Formula databases for high-performance resolution/paramodulation systems. *Journal of Automated Reasoning*, 12:139–156, 1994.
2. C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics. Academic Press, New York, New York, 1973.
3. P. Graf. Substitution tree indexing. In *6th International Conference on Rewriting Techniques and Applications RTA-95*, pages 117–131. Springer LNCS 914, 1995.
4. P. Graf. *Term Indexing.* Springer LNAI 1053, 1996.
5. J.D. McCharen, R. Overbeek, and L. Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.
6. W. McCune. Experiments with discrimination-tree indexing and path-indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992.
7. C. Meyer. Parallel Unit Resulting Resolution. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 1996. http://www.mpi-sb.mpg.de/papers/masters_theses/meyer.ps.gz.
8. H.J. Ohlbach. Abstraction tree indexing for terms. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 479–484. Pitman Publishing, London, August 1990.
9. J.A. Robinson. Automated deduction with hyper-resolution. *International Journal of Comp. Mathematics*, 1:227–234, 1965.
10. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
11. L. Wos. Note on McCune's article on discrimination trees. *Journal of Automated Reasoning*, 9(2):145–146, 1992.