# Substitution Tree Indexing

Peter Graf

MPI–I–94–251 October 1994

## Author's Address

Peter Graf (graf@mpi-sb.mpg.de),
Max–Planck–Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany

## Publication Notes

This report has been submitted for publication elsewhere and will be copyrighted if accepted.

## Acknowledgements

Thanks to Hans Jürgen Ohlbach for his comments on earlier versions of this report. I also like to thank William McCune for providing a part of the term sets.

This work was supported by the "Schwerpunkt Deduktion (Projekt: EDDS)" of the German Science Foundation (DFG).

## Abstract

The performance of a theorem prover crucially depends on the speed of the basic retrieval operations, such as finding terms that are unifiable with (instances of, or more general than) a given query term. In this paper a new indexing method is presented, which outperforms traditional methods such as path indexing, discrimination tree indexing and abstraction trees. Additionally, the new index not only supports term indexing but also provides maintenance and efficient retrieval of substitutions. As confirmed in multiple experiments, substitution trees combine maximal search speed and minimal memory requirements.

# Contents

# 1 Introduction

Searching for a proof, most automatic reasoning systems will accumulate enormous amounts of data – terms, substitutions, clauses, and sometimes also formulae [LO80]. These data must be stored and accessed in various ways. As in standard database technology, *indexing* is the key to *efficiently* retrieving data from large databases.

The structure of logical data, e.g. the structure of terms, is much more complicated than the structure of data stored in a relational database. Furthermore, queries to a logical database [BO94] are also more complex than queries to standard databases.

Typical queries which arise in the context of resolution theorem proving [Rob65] are: Given a database $D$ containing terms (literals) and a query term $t$, find all terms in $D$ which are unifiable with, instances of, or more general than $t$. Most deduction systems currently in use or under development employ sophisticated term indexing techniques in order to increase the speed of this access and thereby to accelerate the search as a whole. Hitherto, the most often applied strategies have been path–indexing [Sti89, McC92, Gra92, Gra94], discrimination tree indexing [McC92, Chr93, BCR93], and abstraction tree indexing [Ohl90a, Ohl90b].

In this paper *substitution tree indexing* is presented as a new indexing technique which combines the advantages of discrimination and abstraction tree indexing. Memory requirement and retrieval times being the main criteria for judging an indexing technique, this paper will show that substitution tree indexing is superior to all known strategies in these points. Substitution trees 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. Figure 1 shows an index for the three substitutions $\{u \mapsto f(a,b)\}$, $\{u \mapsto f(y,b)\}$, and $\{u \mapsto f(b,z)\}$ which obviously represents a term index for the terms $f(a,b)$, $f(y,b)$, and $f(b,z)$. As the name already indicates, the labels of substitution tree nodes are substitutions. Each branch in the tree therefore represents a binding chain for variables. Consequently, the substitutions of a branch from the root node down to a particular node can be composed and yield an instance of the root node's substitution.

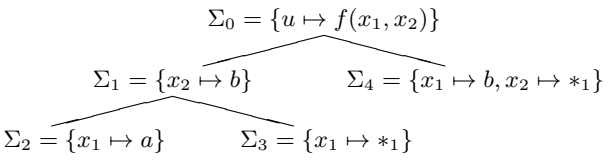



Figure 1: Substitution tree

Before substitutions are inserted into the index, their codomain is renamed. This *normalization* changes all variables in the codomain of a substitution. Renamed variables are called *indicator variables* and are denoted by $*_i$. The substitutions inserted to the index in Fig. 1 therefore were $\{u \mapsto f(a,b)\}$, $\{u \mapsto f(*_1,b)\}$, and $\{u \mapsto f(b,*_1)\}$. This renaming has two main reasons: There is more sharing in the index if the substitutions

are *normalized* and, when searching for instances in the index, indicator variables must not be instantiated.

Consider the substitution $\Sigma = \{u \mapsto f(a,b)\}$ which is represented by the chain of substitutions $\Sigma_0 = \{u \mapsto f(x_1,x_2)\}$, $\Sigma_1 = \{x_2 \mapsto b\}$, and $\Sigma_2 = \{x_1 \mapsto a\}$. The original substitution $\Sigma$ can be reconstructed by simply applying the substitution $\Sigma_2\Sigma_1\Sigma_0$ to $u$. The result of this application is $\Sigma = \{u \mapsto \Sigma_2\Sigma_1\Sigma_0(u)\} = \{u \mapsto \Sigma_2\Sigma_1(f(x_1,x_2))\} = \{u \mapsto \Sigma_2(f(x_1,b))\} = \{u \mapsto f(a,b)\}$.

The retrieval in a substitution tree is based on a backtracking algorithm in addition to an ordinary representation of substitutions as lists of variable–term pairs. This algorithm also needs a backtrackable variable binding mechanism, similar to the one used in PROLOG.

The search for substitutions compatible with $\{u \mapsto f(a,x)\}$ is presented, i.e. substitutions $\tau$ are searched such that $\tau(u)$ is unifiable with $f(a,x)$. We begin with binding the variable $u$ to the term $f(a,x)$ and start the retrieval: The substitution tree is traversed by testing at each node marked with the substitution $\Sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ whether under the current bindings all $x_i$ are unifiable with their appropriate $t_i$. At the root node in our example we unify the terms $f(a,x)$ and $f(x_1,x_2)$ which yields the two bindings $x_1 \mapsto a$ and $x \mapsto x_2$. Consider the first son of the root node marked with $\Sigma_1$ and unify $x_2$ with $b$, because $x_2$ hasn't been bound yet. The resulting binding is $x_2 \mapsto b$ and the leaf node $\Sigma_2$ is the next node to be investigated. As $x_1$ is bound to $a$, the unification problem is trivial and therefore the substitution represented by this leaf node is compatible with $\{u \mapsto f(a,x)\}$. After backtracking node $\Sigma_3$ is found to represent another solution, because the variable $*_1$ is unifiable with $a$. Backtracking deletes the bindings of $*_1$ and $x_2$ and then proceeds with node $\Sigma_4$. Obviously, retrieval can be stopped at this point, because $a$, which is the binding of $x_1$, is not unifiable with $b$.

Substitution trees provide maximal search speed paired with minimal memory requirements. Additionally, they do not only work for term sets but also for sets of substitutions. These substitutions don't even need to have identical domains. The structure is very simple, just a tree of substitutions has to be maintained.

# 2 Preliminaries

## 2.1 Terms, Substitutions, and Unifiers

The standard notions for first order logic are used. Let $V$ and $F$ be two disjoint sets of symbols. $V$ denotes the set of *variable* symbols and $V^* \subset V$ is the set of *indicator variables*. The set of $n$-ary *function* symbols is $F_n$ and $F = \cup F_i$. Furthermore, $T$ is the set of *terms* with $V \subseteq T$ and $f(t_1,\ldots,t_n) \in T$ if $f \in F_n$ and $t_i \in T$. The variables occuring in a term or a set of terms are denoted by $V(t)$. Function symbols with arity 0 are called *constants* . In our examples the symbols $u,v,w,x,y,z \in V$, $*_i \in V^*$, $f,g,h \in F \backslash F_0$ and $a,b,c \in F_0$ are used.

A *substitution* $\sigma$ is a mapping from variables to terms represented by the set of

assignments $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The set $DOM(\sigma) = \{x \in V \,|\, \sigma(x) \neq x\}$ is called *domain* of $\sigma$, the set $COD(\sigma) = \{\sigma(x) \,|\, x \in DOM(\sigma)\}$ the *codomain* of $\sigma$, and $I(\sigma) = V(COD(\sigma))$ is the set of variables *introduced* by $\sigma$. The *composition* 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 $(\sigma\tau)(x) := \sigma(\tau(x))$ for all $x$. It can be computed as $\sigma\tau = \{y_1 \mapsto \sigma(t_1), \ldots, y_m \mapsto \sigma(t_m)\} \cup \{x_i \mapsto s_i \,|\, x_i \in DOM(\sigma) \backslash DOM(\tau)\}$. The *join* of the substitutions $\sigma$ and $\tau$ is defined as $\sigma \bullet \tau := \{y_1 \mapsto \sigma(t_1), \ldots, y_m \mapsto \sigma(t_m)\} \cup \{x_i \mapsto s_i \,|\, x_i \in DOM(\sigma) \backslash I(\tau)\}$. Obviously, for $\sigma = \{x \mapsto a, y \mapsto c\}$ and $\tau = \{z \mapsto f(x)\}$ we have $\sigma\tau = \{z \mapsto f(a), x \mapsto a, y \mapsto c\}$ and $\sigma \bullet \tau = \{z \mapsto f(a), y \mapsto c\}$.

A *unifier* for two terms $s$ and $t$ is a substitution $\sigma$ such that $\sigma(s) = \sigma(t)$. If such a unifier exists $s$ and $t$ are called *unifiable*. Terms may be *non–unifiable* for different reasons. *Clashes* occur when two non-variable symbols are not identical. In contrast to *indirect* clashes and failures resulting from *occur–checks*, a *direct* clash can be detected without considering partial substitutions. A direct clash is detected when unifying $f(a, x)$ and $f(b, y)$, an indirect clash when unifying $f(x, x)$ and $f(a, b)$, and the occur–check detects the failure when unifying $f(x, x)$ and $f(y, g(y))$.

## 2.2 Normalization

A *position* in a term is a finite sequence of natural numbers or the empty position $\varepsilon$. The subterm of a term $t$ at position $p$ is denoted by $t/p$ and $t/\varepsilon = t$. Note that $\varepsilon \circ p = p \circ \varepsilon = p$ with the function $\circ$ representing the concatenation of positions. The set of positions of the term $t = f(t_1, \ldots, t_n)$ is recursively defined by $O(t) = \{\varepsilon\} \cup \{i \circ p \,|\, p \in O(t_i),\, t \notin F_0,\, t \notin V\}$. For example, $O(h(a, g(b), x)) = \{\varepsilon,\, 1,\, 2,\, 2\circ1,\, 3\}$. For constants and variables the set of positions is $\{\varepsilon\}$, In order to properly define the normalization of terms, a total ordering $\prec$ on positions is needed.

**Definition 2.1 (Total Ordering $\prec$ on Positions)** *Let $p = p_1 \circ \ldots \circ p_n$ and $q = q_1 \circ \ldots \circ q_m$ be two different positions with $n, m > 0$. Then*

$$p \prec q \iff \exists i.1 \leq i \leq min(n, m) : p_i < q_i \wedge \forall j.1 \leq j < i : p_j = q_j$$

For example, $1\circ1 \prec 1\circ2$, $1\circ1 \prec 2$, and $1\circ2 \prec 1\circ2\circ1$. Finally, the notion of a normalized term $\overline{s}$ for a term $s$ is introduced. Normalization renames the variables of terms $s$ and $t$ such that for terms equal modulo variable renaming $\overline{s} = \overline{t}$ holds.

**Definition 2.2 (Normalization of Terms)** *Let $s = f(s_1, \ldots, s_n)$ be a term and $F = \{p \,|\, p \in O(s),\, s/p \in V(s),\, \forall q \in O(s).\, q \prec p : s/q \neq s/p\}$ the set of first occurrences of variables in $s$. If $p_1, \ldots, p_m \in F$ and $m = |F|$ and $p_i \prec p_j$ for $1 \leq i < j \leq m$ then the substitution $\sigma = \{s/p_1 \mapsto *_1, \ldots, s/p_m \mapsto *_m\}$ is called normalization and*

$$\overline{s} := \{s/p_1 \mapsto *_1, \ldots, s/p_m \mapsto *_m\}(s)$$

The condition $m = |F|$ in Def. 2.2 ensures that all variables in the term are renamed. For example, $\overline{f(x)} = \overline{f(y)} = f(*_1)$ and $\overline{h(x, x, y)} = \overline{h(z, z, x)} = h(*_1, *_1, *_2)$. The next definition extends the normalization of terms to the normalization of substitutions.

**Definition 2.3 (Normalization of Substitutions)** *Let $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ be a substitution and $f_n \in F_n$ an n–ary function symbol. Additionally, let $<$ be a fixed total ordering on variables and $x_1 < \ldots < x_n$. The normalized substitution $\overline{\sigma}$ is defined as*

$$\overline{\sigma} := \{x_1 \mapsto \overline{f_n(t_1, \ldots, t_n)}/1, \ldots, x_n \mapsto \overline{f_n(t_1, \ldots, t_n)}/n\}$$

For example, if $\sigma = \{x \mapsto f(u, v), y \mapsto f(a, v)\}$ and $x < y$ then $\overline{\sigma} = \{x \mapsto f(*_1, *_2), y \mapsto f(a, *_2)\}$. However, if we had chosen $y < x$ we would have normalized $\sigma$ to $\overline{\sigma} = \{x \mapsto f(*_2, *_1), y \mapsto f(a, *_1)\}$.

# 3 Indexing Techniques Related to Substitution Trees

It has been mentioned, that substitution trees combine features of discrimination and abstraction tree indexing. In this section a brief sketch of these fundamental techniques is presented. Discrimination trees contribute the idea of normalized terms and abstraction trees configurate the terms according to their instance relation. Indexing techniques which don't serve as a perfect filter, like Path–Indexing and some versions of Discrimination tree indexing, are not discussed. These indexing techniques don't check bindings of variables for failures resulting from occurs–check and indirect clashes.

## 3.1 Discrimination Tree Indexing

**Structure.** In this technique the index is a single tree representing the structure of the indexed terms. Pointers to these terms are stored in the leaves of the tree. Each path from the root to a leaf of the discrimination tree corresponds to a set of terms which are equal modulo variable renaming. All these terms are represented by the unique normalized form of the terms. Figure 2 shows a discrimination tree.

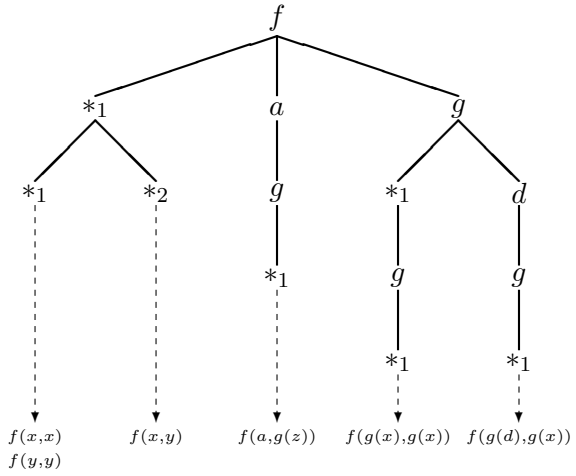



Figure 2: Discrimination tree

**Retrieval.** To answer a query one has to traverse the tree using a backtracking algorithm. Let us take the retrieval of generalizations of a query term as an example: The query term is transformed into a structure which is compatible with the discrimination tree, a so–called *query tree*, by simply inserting the query term into an empty discrimination tree. The backtracking algorithm has to consider that a subterm of the query term can match the $*_i$–nodes and non–variable nodes in the discrimination tree. Both cases must be considered. Additionally, in case of a $*_i$–node the variable $*_i$ is bound to the corresponding subterm of the query term. In case the variable $*_i$ is processed again, the binding of $*_i$ will be checked by testing whether the binding is equal to the current corresponding subterm of the query term.

Finding instances of a query term is also fairly simple: During retrieval of instances, a variable in the query term can match all children of a node in the discrimination tree. However, this time the variable occurs in the query term. Nevertheless, the test whether the bindings of identical variables in the query tree are identical still can't be omitted.

The retrieval of unifiable terms searches for generalizations as well as for instances of subterms. Additionally, an occur check has to be performed and a regular unification routine has to be called in case a $x_i$ has been bound.

**Remarks.** Since discrimination trees are deterministic, they don't depend on the order of term insertion. Insertion of entries is very fast. The memory requirement depends on whether the target terms stored in the index share initial "substrings". In our example there are three terms which end on $g(*_1)$. The whole tree consists of 14 nodes. Abstraction trees consist of less nodes due to a better configuration of the terms.

## 3.2 Abstraction Tree Indexing

**Structure.** Abstraction tree indexing [Ohl90a] exploits the lattice structure of terms. An abstraction tree is based on the usual instance relation on terms which forms a partial ordering.

The tree's nodes are labeled with termlists so that the free variables of the termlists at node $N$ and the termlists of $N$'s subnodes form the domain and codomain of a set of matchers. The abstraction tree in Fig. 3 contains several matchers like $\{x_1 \mapsto y, x_2 \mapsto y\}$ and $\{x_1 \mapsto x_3, x_2 \mapsto g(x_4)\}$ which may be applied to $f(x_1, x_2)$. If all matchers from the root to a leaf of the abstraction tree are applied to the termlist in the root of the tree the resulting term is the one which is represented by this path. Note that abstraction trees are not deterministic. Their structure depends on the order in which terms are inserted.

**Retrieval.** The procedure for accessing unifiable terms takes a node $N$ and a termlist. It unifies the termlist with $N$'s label and applies the unifier to $N$'s variables yielding a new termlist. With each termlist the search for unifiable terms goes down recursively into all subnodes of $N$ until the leaves of the tree are reached.

Generalizations of a query term are found in the same way, except that matching has to be used to prevent instantiation in the query terms.

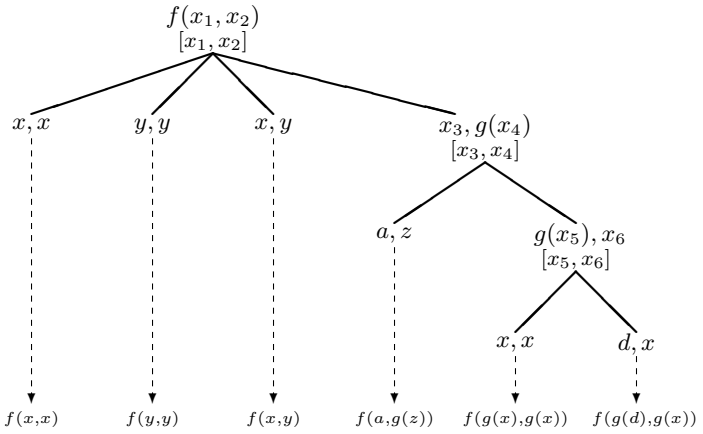



Figure 3: Abstraction tree

Finding instances is also similar to finding unifiable terms. The only difference is that matching is used instead of unification at the leaf nodes. Note that it is not possible to use matching on all nodes.

**Remarks.** Compared to the discrimination tree in Fig. 2 the abstraction tree for the same indexed set consists of 9 nodes only. Additionally, the fact that there are three terms ending on $g(*_1)$ is represented in the tree. However, abstraction tree indexing has some disadvantages: First of all, the trees contain lots of variable renamings which are not necessary. In our example tree the variable $x_1$ is renamed to $x_3$ just to have a consistent tree. Secondly, variables of indexed terms may occur in leaf nodes of the tree only. This implies that an algorithm looking for instances of a query term at an inner node of the tree cannot exploit the fact that a variable in an indexed term must not be instantiated. The consequence is that it will have to use unification instead of matching at inner nodes – and visit more nodes. As a consequence of these disadvantages our abstraction tree contains 16 assignments.

# 4 Substitution Tree Indexing

Substitution trees were developed to increase the performance of indexing. The main difference compared to abstraction trees lies in the representation of variables of indexed substitutions. Additionally, variable renamings are avoided. To this end the structure of the nodes of trees was simplified such that substitutions are stored in contrast to termlists and lists of variables. Variables of indexed terms are represented by indicator variables just like in discrimination trees and may now occur at arbitrary positions in the substitution tree. Figure 4 shows our standard term set. We only need three auxiliary variables and the whole tree contains only 9 assignments in contrast to the 16 assignments of the abstraction tree in Fig. 3. However, the main advantages of abstraction trees are preserved. We can also perform a merge on two trees, for example. Essentially, substitution trees unify the

advantages of abstraction and discrimination trees and result in an outstanding indexing technique.

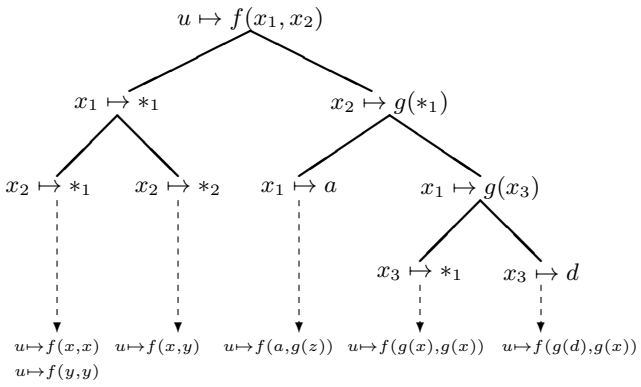



Figure 4: Substitution tree

In our example, we have stored 6 substitutions in the substitution tree. The domains of all these substitutions are identical, but this is not necessary. Substitution trees may also contain substitutions with different domains, which is shown later. We use a backtracking algorithm to find substitutions in the tree with specific properties. All retrieval algorithms are based on backtrackable variable bindings and algorithms for unification and matching which take variable bindings into account. Insertion of a substitution into the index is a complex operation. In contrast to insertion, the deletion of entries is much more straight forward and even complex deletion operations, like the deletion of all compatible substitutions in a substitution tree, can easily be accomplished.

**Definition 4.1 (Substitution Tree)** *A substitution tree is either an empty tree $\varepsilon$ or it can be described by a tuple $(\Sigma, \Omega)$ where $\Sigma$ is a substitution and $\Omega$ is a set of substitution trees. The following two conditions hold:*

1. *A node in the tree is either a leaf node $(\Sigma, \emptyset)$ or an inner node $(\Sigma, \Omega)$ with $|\Omega| \geq 2$.*
2. *For every path $(\Sigma_1, \Omega_1), \ldots, (\Sigma_n, \Omega_n)$ from the root to a leaf of a non–empty tree we have*
   (a) *$I(\Sigma_n \bullet \ldots \bullet \Sigma_1) \subset V^*$.*
   (b) *$DOM(\Sigma_i) \cap (DOM(\Sigma_0) \cup \ldots \cup DOM(\Sigma_{i-1})) = \emptyset$.*

*Additionally, $Subst(N)$ denotes the set of all substitutions contained in the substitution tree $N$. We define*

$$Subst((\Sigma, \Omega)) := \{\Sigma\} \cup \bigcup_{N' \in \Omega} Subst(N').$$

In the following we will need the definition of variables which are *open* at a specific node in the tree.

**Definition 4.2 (Open and Closed Variables)** *A variable $x$ is called* open *at node $N_n$ in a substitution tree if $(\Sigma_0, \Omega_0)$, ..., $(\Sigma_n, \Omega_n)$ is the path from the root of the tree to node $N_n$ and*

$$x \in (\bigcup_{0 \le i \le n} I(\Sigma_i) \quad \setminus \quad \bigcup_{0 \le i \le n} DOM(\Sigma_i)).$$

*Variables which are not open are called* closed *.*

The condition $I(\Sigma_n \bullet \ldots \bullet \Sigma_1) \subset V^*$ in Def. 4.1 implies that all non–indicator variables are closed at leaf nodes in the tree.

# 5 Standard Retrieval

As stated in the introduction, the retrieval of substitutions providing specific properties is very important in the field of theorem proving. In this section a general approach to retrieval in substitution trees is presented. Such a retrieval is able to support the search for complementary literals in a large set of clauses for resolution, for example. Additionally, indexing can be used to support subsumption [WOL91, Vor94]. The retrieval of compatible substitutions as well as the search for instances or generalizations of substitutions have a very important feature in common: The search in the index is started for a single substitution which in this context is called *query*. In contrast to the merge which is introduced in Section 6 such a retrieval is called *standard retrieval*.

## 5.1 Theoretical Foundations

Retrieval in substitution trees is very simple. Generally, the retrieval algorithm checks each node of the tree for some special conditions. If the conditions are fulfilled the algorithm proceeds with the subnodes of the node that has been successfully tested. On its way down to the leaf nodes of the tree, the set of passed nodes is collected.

There are three different tests which have to be performed: Find more general substitutions, compatible substitutions, and instances. The functions $\mathcal{G}$, $\mathcal{I}$, and $\mathcal{U}$ support the tests at the nodes of the tree. For each assignment $x_i \mapsto t_i$ of the current node's substitution the functions test whether the variable $x_i$ or whatever it is bound to is more general, an instance of, or unifiable with $t_i$. Each of these functions can be used as a parameter for the retrieval function *search*.

**Definition 5.1 (Substitution Functions)** *Let $N = (\Sigma, \Omega)$ be a substitution tree and $\rho$ a substitution. Then*

$$\begin{aligned}
\mathcal{G}(N, \rho) &:= \{\, \sigma \mid \forall x_i \in DOM(\Sigma).\ \sigma\rho\Sigma(x_i) = \rho(x_i) \text{ and } \sigma \text{ is most general} \} \\
\mathcal{U}(N, \rho) &:= \{\, \sigma \mid \forall x_i \in DOM(\Sigma).\ \sigma\rho\Sigma(x_i) = \sigma\rho(x_i) \text{ and } \sigma \text{ is most general} \} \\
\mathcal{I}(N, \rho) &:= \{\, \sigma \mid \sigma \in \mathcal{U}(N, \rho) \text{ and } DOM(\sigma) \cap V^* = \emptyset \}
\end{aligned}$$

The retrieval function *search* is defined. It takes the substitution $\Sigma$ which is stored at node $N = (\Sigma, \Omega)$ in the tree and tests $\Sigma$ against the current variable bindings $\rho$ using one

of the substitution functions. Although, one might only be interested in leaf nodes found in the substitution tree, the definition of *search* will produce a set of nodes which have successfully passed the test, no matter if they are leaf nodes or not.

**Definition 5.2 (Retrieval Function)** *Let $N = (\Sigma, \Omega)$ be a node in a substitution tree, $\rho$ a substitution, and $X$ one of the functions $\mathcal{U}$, $\mathcal{I}$, or $\mathcal{G}$. The retrieval function search is recursively defined as*

$$search(N, \rho, X) := \begin{cases} \{N\} \cup \bigcup_{N' \in \Omega} search(N', \sigma\rho, X) & \text{if } \exists \sigma \in X(N, \rho) \\ \emptyset & \text{otherwise} \end{cases}$$

The next lemma is essential for the soundness of the retrieval operation for compatible substitutions. It states, that two unifiable terms remain unifiable even if a substitution is applied to one of them. However, this substitution has to fulfill some extra conditions.

**Lemma 5.1** *Let $s$ and $t$ be two unifiable terms, $\rho$ an idempotent most general unifier such that $\rho(s) = \rho(t)$, $\Sigma$ a substitution with $DOM(\Sigma) \cap V(t) = \emptyset$, and $\exists \tau \forall x \in DOM(\Sigma).\, \tau\rho\Sigma(x) = \tau\rho(x)$. Then*

$$\tau\rho\Sigma(s) = \tau\rho(t)$$

**Proof:** As the terms $s$ and $t$ are unifiable by applying the unifier $\rho = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, we have to show that for all assignments $x_i \mapsto \rho(x_i)$ of $\rho$ the equation $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$ holds. Then $\tau\rho\Sigma(s) = \tau\rho\Sigma(t)$ and as $DOM(\Sigma) \cap V(t) = \emptyset$ we have $\tau\rho\Sigma(s) = \tau\rho(t)$. In order to prove that $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$, we have to distinguish four cases:

**Case 1:** $(DOM(\Sigma) \cap V(\rho(x_i)) = \emptyset \ \wedge \ x_i \in DOM(\Sigma))$
As $x_i \in DOM(\Sigma)$ we use the precondition $\exists \tau \forall x \in DOM(\Sigma).\, \tau\rho\Sigma(x) = \tau\rho(x)$ to simplify the equation $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$ to $\tau\rho(x_i) = \tau\rho\Sigma\rho(x_i)$. As $DOM(\Sigma) \cap V(\rho(x_i)) = \emptyset$ this is equivalent to $\tau\rho(x_i) = \tau\rho\rho(x_i)$, which holds, because $\rho$ is idempotent.

**Case 2:** $(DOM(\Sigma) \cap V(\rho(x_i)) = \emptyset \ \wedge \ x_i \notin DOM(\Sigma))$
As $x_i \notin DOM(\Sigma)$ we can simplify each occurrence of $\Sigma(x_i)$ to $x_i$. Therefore the equation $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$ also simplifies to $\tau\rho(x_i) = \tau\rho\Sigma\rho(x_i)$. Proceed as in Case 1.

**Case 3:** $(DOM(\Sigma) \cap V(\rho(x_i)) \neq \emptyset \ \wedge \ x_i \in DOM(\Sigma))$
The equation $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$ simplifies to $\tau\rho(x_i) = \tau\rho\Sigma\rho(x_i)$, if the precondition $\exists \tau \forall x \in DOM(\Sigma).\, \tau\rho\Sigma(x) = \tau\rho(x)$ is used. The equation $\tau\rho(x_i) = \tau\rho\Sigma\rho(x_i)$ is equivalent to $\tau\rho\rho(x_i) = \tau\rho\Sigma\rho(x_i)$, because $\rho$ is idempotent. We consider the variables $y_j$ in $\rho(x_i)$. Again we distinguish two cases: First, if $y_j \in DOM(\Sigma)$ we may apply the precondition $\exists \tau \forall x \in DOM(\Sigma).\, \tau\rho\Sigma(x) = \tau\rho(x)$ to the variables $y_j$ in $\rho(x_i)$ and get $\tau\rho(x_i) = \tau\rho(x_i)$. Second, if $y_j \notin DOM(\Sigma)$ the term $\Sigma\rho(x_i)$ simplifies to $\rho(x_i)$ and therefore $\tau\rho\rho(x_i) = \tau\rho\rho(x_i)$. By induction on the structure of the terms the equation $\tau\rho\rho(x_i) = \tau\rho\Sigma\rho(x_i)$ holds.

**Case 4:** $(DOM(\Sigma) \cap V(\rho(x_i)) \neq \emptyset \ \wedge \ x_i \notin DOM(\Sigma))$
Again, we can simplify each occurrence of $\Sigma(x_i)$ to $x_i$, because $x_i \notin DOM(\Sigma)$. Therefore the equation $\tau\rho\Sigma(x_i) = \tau\rho\Sigma\rho(x_i)$ also simplifies to $\tau\rho(x_i) = \tau\rho\Sigma\rho(x_i)$. Proceed as in Case 3. □

**Theorem 5.1 (Compatible Substitutions)** *Let $N_0, \ldots, N_n$ be a path from the root $N_0 = (\Sigma_0, \Omega_0)$ to a node $N_n = (\Sigma_n, \Omega_n)$ of a substitution tree and $\varphi$ a query substitution. Then*

$$N_n \in search(N_0, \varphi, \mathcal{U}) \iff \exists\sigma\forall x \in V. \ \ \sigma\Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$$

**Proof:** Soundness is shown by induction on depth of the node in the tree and completeness is proved by contradiction.

**⇒:** $\underline{n = 0:}$ In case $N_0$ is a root node and $N_0 \in search(N_0, \varphi, \mathcal{U})$ the definition of *search* implies $\exists\sigma = \mathcal{U}(N_0, \varphi)$. The definition of $U$ yields

$$\exists\sigma\forall x \in DOM(\Sigma_0).\ \sigma\varphi\Sigma_0(x) = \sigma\varphi(x)$$

In substitution trees we always have $DOM(\varphi) \cap I(\Sigma_0) = \emptyset$ and therefore

$$\exists\sigma\forall x \in V.\ \sigma\Sigma_0(x) = \sigma\varphi(x)$$

$\underline{n > 0:}$ In case $N_{n+1} \in search(N_0, \varphi, \mathcal{U})$ and $N_{n+1} \neq N_0$ we can directly conclude from the definition of *search* that for the father node $N_n$ of $N_{n+1}$ we have $N_n \in search(N_0, \varphi, \mathcal{U})$ and by the induction hypothesis

$$\exists\rho\forall x \in V.\ \rho\Sigma_n \ldots \Sigma_0(x) = \rho\varphi(x)$$

As $N_{n+1} \in search(N_0, \varphi, \mathcal{U})$ we can conclude that $\exists\tau = \mathcal{U}(N_{n+1}, \rho)$ which is equivalent to

$$\exists\tau\forall x_i \in DOM(\Sigma_{n+1}).\ \tau\rho\Sigma_{n+1}(x_i) = \tau\rho(x_i)$$

Application of Lemma 5.1 to the induction hypothesis yields

$$\exists\rho,\tau\forall x \in V.\ \tau\rho\Sigma_{n+1} \ldots \Sigma_0(x) = \tau\rho\varphi(x)$$

The induction is finished by setting $\sigma = \tau\rho$ which yields

$$\exists\sigma\forall x \in V.\ \sigma\Sigma_{n+1} \ldots \Sigma_0(x) = \sigma\varphi(x)$$

**⇐:** Assume that $N_n \notin search(N_0, \varphi, \mathcal{U})$ and $\exists\sigma\forall x \in V. \ \ \sigma\Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$. Then there is a path $N_0, \ldots, N_{i-1}, N_i, \ldots N_n$ such that either $N_i \notin search(N_0, \varphi, \mathcal{U})$ and $\forall j.\ 0 \leq j < i :\ N_j \in search(N_0, \varphi, \mathcal{U})$ or $N_n$ is the root node. The case that $N_n$ is the root node can directly be excluded as it contradicts the definition of $\mathcal{U}$. We consider the first case: $N_i \notin search(N_0, \varphi, \mathcal{U})$ and $\forall j.\ 0 \leq j < i :\ \ N_j \in search(N_0, \varphi, \mathcal{U})$ implies that

$$\mathcal{U}(N_i, \sigma_{i-1} \ldots \sigma_0\varphi) = \emptyset$$

This is equivalent to $\mathcal{U}(N_i, \sigma\varphi) = \emptyset$, because one can set $\sigma_i = \sigma$ and for idempotent $\sigma$ we have $\sigma\sigma = \sigma$. Thus,

$$\neg\exists\sigma\forall x \in DOM(\Sigma_i).\ \sigma\varphi\Sigma_i(x) = \sigma\varphi(x)$$

which yields

$$\neg\exists\sigma\forall x \in DOM(\Sigma_i).\ \sigma\varphi\Sigma_i \ldots \Sigma_0(x) = \sigma\varphi(x)$$

because the definition of substitution trees states that $DOM(\Sigma_i) \cap (DOM(\Sigma_0) \cup \ldots \cup DOM(\Sigma_{i-1})) = \emptyset$. Additionally, $DOM(\varphi) \cap I(\Sigma_i) = \emptyset$ and therefore

$$\neg\exists\sigma\forall x \in DOM(\Sigma_i).\ \sigma\Sigma_i \ldots \Sigma_0(x) = \sigma\varphi(x)$$

As the variables $x \in DOM(\Sigma_i)$ don't occur in the substitutions of the nodes $N_{i+1}, \ldots N_n$ this is equivalent to

$$\neg\exists\sigma\forall x \in DOM(\Sigma_i).\ \sigma\Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$$

which contradicts our assumptions. □

The proof of the soundness and completeness of the retrieval of more general substitutions is very similar to the one for compatible substitutions. Again a lemma is presented first.

**Lemma 5.2** *Let $s$ and $t$ be two terms, $\rho$ a matcher such that $\rho(s) = t$, $\Sigma$ a substitution with $DOM(\Sigma) \cap V(t) = \emptyset$, and $\exists\tau\forall x \in DOM(\Sigma).\ \tau\rho\Sigma(x) = \rho(x)$. Then*

$$\tau\rho\Sigma(s) = t$$

**Proof:** It has to be shown that for all variables $x \in V(s)$ the condition $\tau\rho\Sigma(x) = \rho(x)$ holds, because $\rho$ doesn't instantiate $t$ and $DOM(\Sigma) \cap V(t) = \emptyset$. Two different cases have to be considered. In the first case $x \in DOM(\Sigma)$ we can use the condition $\exists\tau\forall x \in DOM(\Sigma).\ \tau\rho\Sigma(x) = \rho(x)$ to prove $\tau\rho\Sigma(x) = \rho(x)$. In the second case $x \notin DOM(\Sigma)$ which implies that $\Sigma(x)$ simplifies to $x$. However, the equation $\tau\rho(x) = \rho(x)$ is fulfilled if $\tau$ is set to the empty substitution. □

**Theorem 5.2 (More General Substitutions)** *Let $N_0, \ldots, N_n$ be a path from the root $N_0 = (\Sigma_0, \Omega_0)$ to a node $N_n = (\Sigma_n, \Omega_n)$ of a substitution tree and $\varphi$ a query substitution. Then*

$$N_n \in search(N_0, \varphi, \mathcal{G}) \iff \exists\sigma\forall x \in V.\ \ \sigma\Sigma_n \ldots \Sigma_0(x) = \varphi(x)$$

**Proof:** Using Lemma 5.2 the theorem can be proved in analogy to Theorem 5.1. □

The definitions of the functions $\mathcal{U}$ and $\mathcal{I}$ are identical except for $\mathcal{I}$ doesn't bind indicator variables. This fact is employed by the next theorem.

**Theorem 5.3 (Instances)** *Let $N_0, \ldots, N_n$ be a path from the root $N_0 = (\Sigma_0, \Omega_0)$ to a leaf node $N_n = (\Sigma_n, \Omega_n)$ of a substitution tree and $\varphi$ a query substitution. Then*

$$N_n \in search(N_0, \varphi, \mathcal{I}) \iff \exists\sigma\forall x \in V.\ \ \Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$$

**Proof:** We have $N_n \in search(N_0, \varphi, \mathcal{I})$ if and only if $N_n \in search(N_0, \varphi, \mathcal{U})$ and $DOM(\sigma) \cap V^* = \emptyset$ where $\exists\sigma\forall x \in V.\ \ \sigma\Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$ if and only if $\exists\sigma\forall x \in V.\ \ \Sigma_n \ldots \Sigma_0(x) = \sigma\varphi(x)$ and $DOM(\sigma) \cap V^* = \emptyset$, because the definition of substitution trees contains $I(\Sigma_n \bullet \ldots \bullet \Sigma_0) \subset V^*$. □

Note that the theorem is valid only for leaf nodes $N_n$.

## 5.2 Implementing Standard Retrieval

Now that soundness and correctness of the definition of the retrieval have been shown, we can describe an efficient implementation. In our implementations we used the following technique: The algorithm is based on a stack of variable bindings. Variables are pushed on the stack by the function `unify(`*N*`,`*STACK*`,`*BINDINGS*`)`, for example. This function checks for each assignment $x_i \mapsto t_i$ of $N$'s substitution $\Sigma = \{\ldots, x_i \mapsto t_i, \ldots\}$ whether $x_i$ is unifiable with $t_i$. The bindings of variables to terms which are necessary to make the terms identical are pushed on the *STACK* and counted in *BINDINGS*. Obviously, this unification has to consider variable bindings in the terms to be unified. All occurrences of a variable can easily be changed if the variable is stored just once and shared in all occurrences. Additionally, the function `backtrack(`*STACK*, *BINDINGS*`)` pops *BINDINGS* bindings from the *STACK*. Therefore, this function resets the stack to the state before the unification. A retrieval algorithm based on these functions is presented in Fig. 5.

```
algorithm search (input N, STACK; output HITS)
begin
   HITS = ∅;
   if (unify(N, STACK, BINDINGS))
      HITS = HITS ∪{N};
      forall (N' ∈ Ω^N)
         HITS = HITS ∪ search(N', STACK);
   backtrack(STACK, BINDINGS);
   return HITS;
end;
```

Figure 5: Algorithm for retrieval function *search*

## 5.3 Example

Figure 6 depicts a sequence of stacks resulting from the search for substitutions compatible with $\{u \mapsto f(a, y)\}$. Originally, the stack is empty. Before we start the retrieval algorithm, all variables in the domain of the query substitution are bound to their corresponding codomain, i.e. the bindings are pushed on the stack (compare stack "Init"). The recursive retrieval algorithm starts with the root node. In case it succeeds, the resulting stack is marked with "Success". If the corresponding node in the tree is a leaf node, "**Success**" is written boldface.

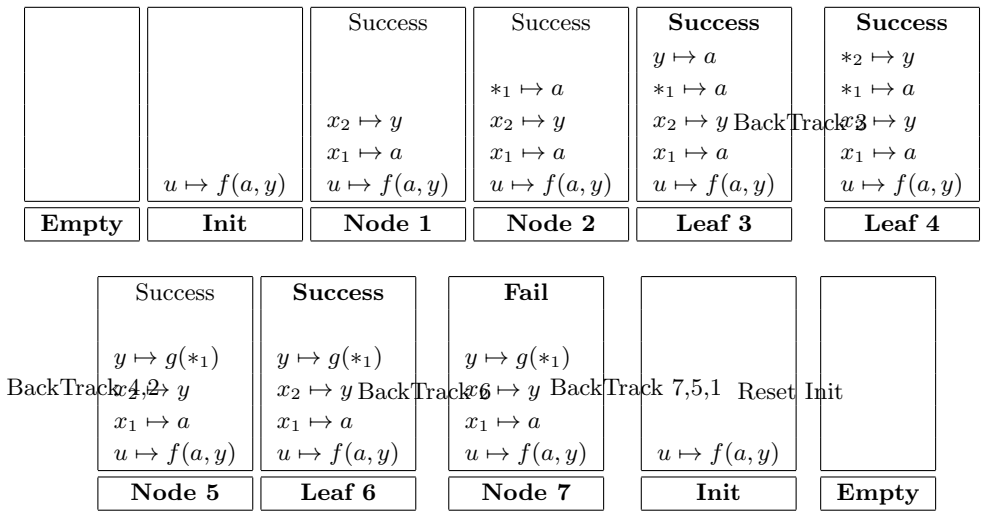



Figure 6: Stack of bindings during retrieval for $\{u \mapsto f(a,y)\}$

In the example the three leaf nodes 3, 4, and 6 are found. They correspond to the substitutions $\{u \mapsto f(x,x)\}$, $\{u \mapsto f(y,y)\}$,$\{u \mapsto f(x,y)\}$, and $\{u \mapsto f(a,g(z))\}$. Due to the renaming of the variables in the substitutions inserted the retrieval in substitution trees does not identify the variable $y$ in the query substitution $\{u \mapsto f(a,y)\}$ with the $y$ occurring in the inserted substitutions.

# 6 Merge

An advanced operation on substitution trees is merging. The merge computes the set of compatible substitutions stored in two different trees. Substitutions are compatible if the codomains of identical variables in the two substitutions are simultaneously unifiable. For example, the substitutions $\{x \mapsto f(u,b), z \mapsto h(w)\}$ and $\{x \mapsto f(a,v), y \mapsto g(v)\}$ are compatible and the result of the merge of the two substitutions is $\{x \mapsto f(a,b), y \mapsto g(b), z \mapsto h(w)\}$.

An application of the merge operation is hyperresolution. In this context a simultaneous unifier has to be found for a set of literals. To this end we maintain a substitution tree for each literal of the nucleus which contains all unifiers with literals of the electrons. By merging the substitution trees attached to the literals of the nucleus a simultaneous unifier can be computed efficiently.

## 6.1 Theoretical Foundations

The merge operation is very similar to an ordinary retrieval except that two trees are traversed in parallel. To this end we alternately test nodes in both of the trees.

**Definition 6.1 (Merge)** *Let $M = (\Sigma^M, \Omega^M)$ and $N = (\Sigma^N, \Omega^N)$ be two substitution trees, $\rho$ a substitution. The retrieval function merge is recursively defined as follows:*

$$merge(M, N, \rho) := \begin{cases} \{(M, N') \mid N' \in search(N, \sigma\rho, \mathcal{U})\} \\ \qquad \text{if } \Omega^M = \emptyset \text{ and } \Omega^N \neq \emptyset \text{ and } \exists\sigma \in \mathcal{U}(M, \rho) \\ \{(M', N) \mid M' \in search(M, \sigma\rho, \mathcal{U})\} \\ \qquad \text{if } \Omega^M \neq \emptyset \text{ and } \Omega^N = \emptyset \text{ and } \exists\sigma \in \mathcal{U}(N, \rho) \\ \{(M, N)\} \cup \bigcup_{M' \in \Omega^M} \bigcup_{N' \in \Omega^N} merge(M', N', \tau\sigma\rho) \\ \qquad \text{if } \exists\sigma \in \mathcal{U}(M, \rho) \text{ and } \exists\tau \in \mathcal{U}(N, \sigma\rho) \end{cases}$$

**Theorem 6.1 (*merge* is Sound and Complete)** *Let $M_0, \ldots, M_m$ be a path from the root $M_0 = (\Sigma_0^M, \Omega_0^M)$ to a node $M_m = (\Sigma_m^M, \Omega_m^M)$ of a substitution tree and $N_0, \ldots, N_n$ be a path from the root $N_0 = (\Sigma_0^N, \Omega_0^N)$ to a node $N_n = (\Sigma_n^N, \Omega_n^N)$ of another tree. Additionally, $I(Subst(M_0)) \cap I(Subst(N_0)) = \emptyset$. Then*

$$(M_m, N_n) \in merge(M_0, N_0, \emptyset) \iff \exists\sigma\forall x \in V.\ \ \sigma\Sigma_m^M \ldots \Sigma_0^M(x) = \sigma\Sigma_n^N \ldots \Sigma_0^N(x)$$

**Proof:** We proof soundness by nested induction on $m$ and $n$. Completeness is shown by contradiction.

**⇒:** $\underline{m = 0}$:

$\underline{n = 0}$: We have $(M_0, N_0) \in merge(M_0, N_0, \emptyset)$ if $\exists\rho \in \mathcal{U}(M_0, \emptyset)$ and $\exists\tau \in \mathcal{U}(N_0, \rho)$. $\exists\rho \in \mathcal{U}(M_0, \emptyset)$ implies $\rho = \Sigma_0^M$. As $\exists\tau \in \mathcal{U}(N_0, \Sigma_0^M)$ we have

$$\exists\sigma\forall x \in V.\ \ \sigma\Sigma_0^M(x) = \sigma\Sigma_0^N(x)$$

$\underline{n > 0}$: We have $(M_0, N_n) \in merge(M_0, N_0, \emptyset)$ if $\exists\rho \in \mathcal{U}(M_0, \emptyset)$ and additionally $N_n \in search(N_0, \rho, \mathcal{U})$. Again $\exists\rho \in \mathcal{U}(M_0, \emptyset)$ implies $\rho = \Sigma_0^M$. Theorem 5.1 yields the equivalence of $N_n \in search(N_0, \Sigma_0^M, \mathcal{U})$ and

$$\exists\sigma\forall x \in V.\ \sigma\Sigma_0^M(x) = \sigma\Sigma_n^N \ldots \Sigma_0^N(x)$$

$\underline{m > 0}$:

$\underline{n = 0}$: We have $(M_m, N_0) \in merge(M_0, N_0, \emptyset)$ if $\exists\rho \in \mathcal{U}(N_0, \emptyset)$ and additionally $M_m \in search(M_0, \rho, \mathcal{U})$. Again $\exists\rho \in \mathcal{U}(N_0, \emptyset)$ implies $\rho = \Sigma_0^N$. Theorem 5.1 yields the equivalence of $M_m \in search(M_0, \Sigma_0^N, \mathcal{U})$ and

$$\exists\sigma\forall x \in V.\ \sigma\Sigma_m^M \ldots \Sigma_0^M(x) = \sigma\Sigma_0^N(x)$$

$\underline{n > 0}$: The fact that $(M_{m+1}, N_{n+1}) \in merge(M_0, N_0, \emptyset)$ implies that the father nodes $(M_m, N_n) \in merge(M_0, N_0, \emptyset)$. The induction hypothesis therefore yields $\exists\rho\forall x \in V.\ \rho\Sigma_m^M \ldots \Sigma_0^M(x) = \rho\Sigma_n^N \ldots \Sigma_0^N(x)$. Additionally, $(M_{m+1}, N_{n+1}) \in merge(M_0, N_0, \emptyset)$ implies that $\exists\rho' \in \mathcal{U}(M_{m+1}, \rho)$ and $\exists\tau \in \mathcal{U}(N_{n+1}, \rho'\rho)$. We apply Lemma 5.1 twice to the induction hypothesis and finally get

$$\exists\sigma\forall x \in V.\ \sigma\Sigma_{m+1}^M \ldots \Sigma_0^M(x) = \sigma\Sigma_{n+1}^N \ldots \Sigma_0^N(x)$$

$\Leftarrow$: Assume that $(M_m, N_n) \notin merge(M_0, N_0, \emptyset)$ and

$$\exists\sigma\forall x \in V.\ \ \sigma\Sigma_m^M \ldots \Sigma_0^M(x) = \sigma\Sigma_n^N \ldots \Sigma_0^N(x).$$

Then there is a path $M_0, \ldots, M_{i_{j-1}}, M_{i_j}, \ldots M_m$ in the tree $M_0$ and a path $N_0, \ldots, N_{i_{j-1}}, N_{i_j}, \ldots N_n$ in the tree $N_0$ such that $(M_{i_j}, N_{i_j}) \notin merge(M_0, N_0, \emptyset)$ and $(M_{i_{j-1}}, N_{i_{j-1}}) \in merge(M_0, N_0, \emptyset)$. This implies that either $\mathcal{U}(M_{i_j}, \sigma) = \emptyset$ or $\mathcal{U}(N_{i_j}, \sigma) = \emptyset$.

Case 1: $(\mathcal{U}(M_{i_j}, \sigma) = \emptyset)$
We have

$$\neg\exists\sigma\forall x \in DOM(\Sigma_{i_j}^M).\ \sigma\Sigma_{i_j}^M(x) = \sigma(x)$$

which is equivalent to

$$\neg\exists\sigma\forall x \in DOM(\Sigma_{i_j}^M).\ \sigma\Sigma_m^M \ldots \Sigma_0^M(x) = \Sigma_n^N \ldots \Sigma_0^N\sigma(x)$$

as $I(Subst(M_0)) \cap I(Subst(N_0)) = \emptyset$ and $DOM(\Sigma_{i_j}^M) \cap (DOM(\Sigma_0^M) \cup \ldots \cup DOM(\Sigma_{i_{j-1}}^M)) = \emptyset$ in substitution trees.

Case 2: $(\mathcal{U}(N_{i_j}, \sigma) = \emptyset)$
Then

$$\neg\exists\sigma\forall x \in DOM(\Sigma_{i_j}^N).\ \sigma\Sigma_{i_j}^N(x) = \sigma(x)$$

which corresponds to

$$\neg\exists\sigma\forall x \in DOM(\Sigma_{i_j}^N).\ \sigma\Sigma_m^M \ldots \Sigma_0^M(x) = \Sigma_n^N \ldots \Sigma_0^N\sigma(x)$$

for analogous reasons.

In both cases we have a contradiction to our assumptions. □

## 6.2 Implementing the Merge

An efficient implementation for the merge of two substitution trees is presented. It is based on the same functions and ideas as the algorithm for standard retrieval. The algorithm presented in Fig. 7 has some very special features. First of all, the substitutions of $M$ are tested before each call of the function `merge(`$M$`,`$N$`,`$STACK$`)`. This approach minimizes backtracking on bindings made in context with node $M$. As a consequence, however, the first unification of the root of $M$ has to be done before `merge` is called. Second, the roles of the trees $M$ and $N$ are swapped in case $N$ is a leaf node and $M$ is not. Third, in case one of the trees is a leaf node, the function `merge` does a retrieval identical to the one performed by `search`.

In Def. 6.1 the result of the merge operation is a set of tuples. In many applications, however, the user is interested in the common instances which are produced during unification of the substitutions. These instances can easily be stored in another substitution tree which then represents the result of the merge. The procedure in order to compute this resulting tree is straight forward: The common instance found by the merge is normalized and inserted. Note that the variable bindings done in context with the renaming have to be reset before the merge is continued.

```
algorithm merge (input M, N, STACK; output HITS)
begin
  HITS = ∅;
  if (unify(N, STACK, BINDINGS^N))
    HITS = HITS ∪ {(M, N)};
    if (is_leaf(M) and not is_leaf(N))
      forall (N' ∈ Ω^N)
        HITS = HITS ∪ merge(M, N',STACK);
    else if (is_leaf(N) and not is_leaf(M))
      forall (M' ∈ Ω^M)
        HITS = HITS ∪ merge(N, M',STACK);
    else
      forall (M' ∈ Ω^M)
        if (unify(M', STACK, BINDINGS^M))
          forall (N' ∈ Ω^M)
            HITS = HITS ∪ merge(M', N',STACK);
        backtrack(STACK, BINDINGS^M);
  backtrack(STACK, BINDINGS^N);
  return HITS;
end;
```

Figure 7: Algorithm for retrieval function *merge*

## 6.3 Example

This example shows how an implementation of the *merge* function computes all pairs of compatible substitutions in the two substitution trees depicted in Fig. 8.

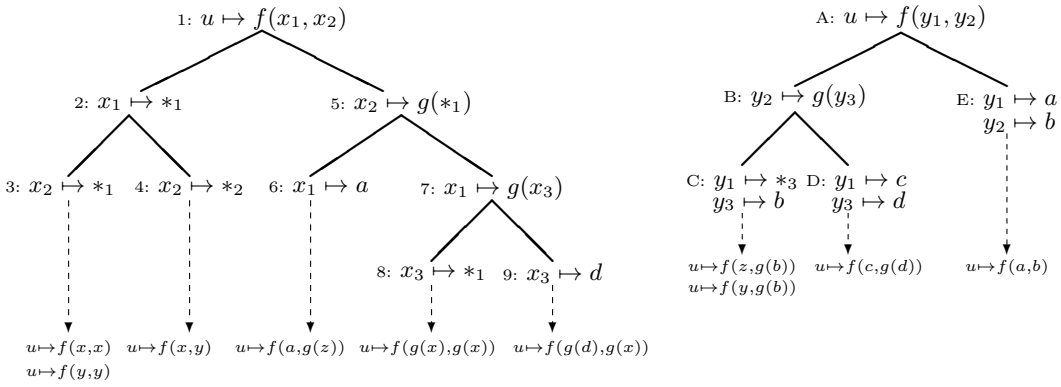



Figure 8: Trees to be merged

Note that all variables introduced in the codomains of the substitutions in the trees are disjoint. The tree in Fig. 4 uses the variables $x_i$ and the indicator variables $*_1$ and $*_2$. The tree in Fig. 8 uses $y_i$ and the indicator variable $*_3$.

The merge in our example finds the following compatible substitutions: The substitutions represented by leaf 3 are compatible with the substitutions represented by leaf $C$.

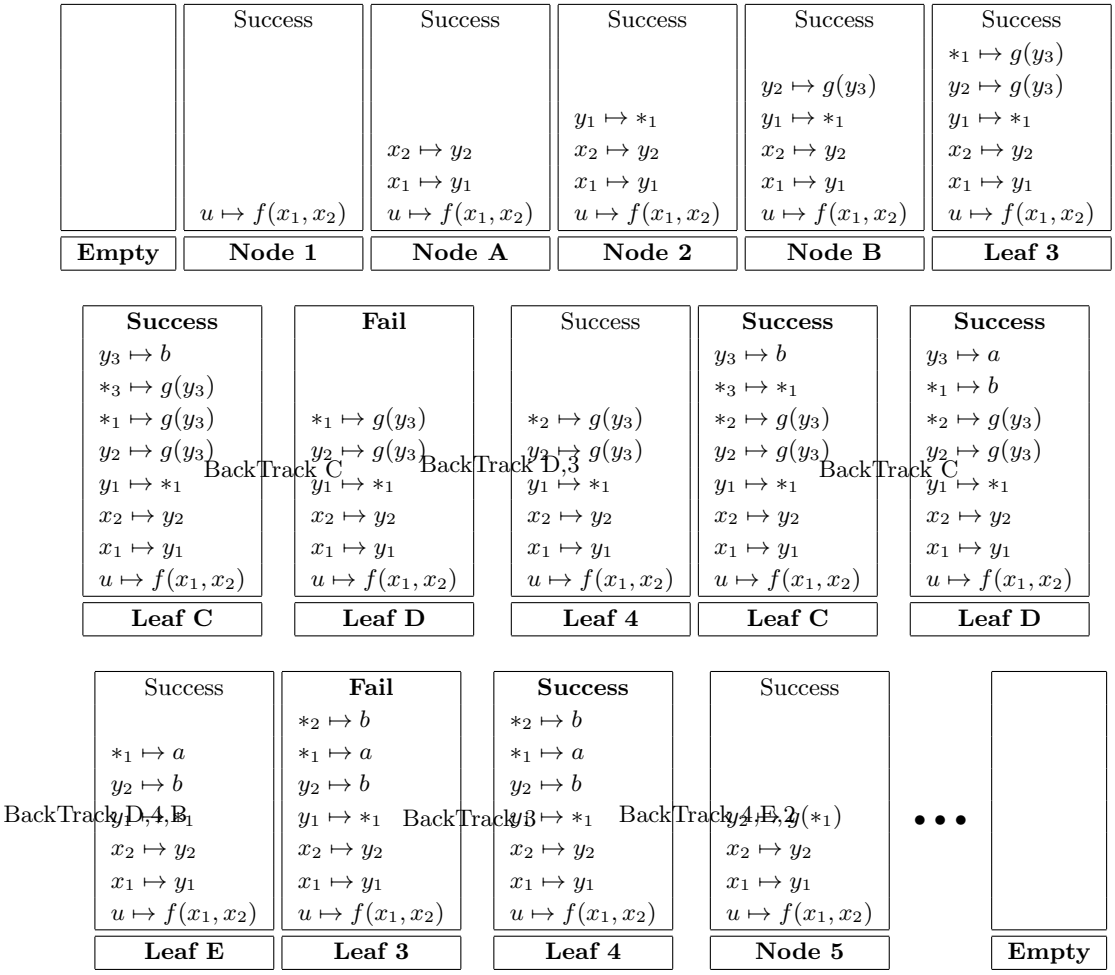



Figure 9: Stack of bindings during merge of two substitution trees

Additionally, we have the pairs 4–$C$, 4–$D$, 4–$E$, 6–$C$, 8–$C$, and 9–$D$.

# 7 Insertion

The insertion of new entries to a substitution tree is more difficult than the retrieval or the deletion of entries. In this section the insertion function and the central notion of the *most specific common generalization* are introduced. These generalizations are needed if the insertion of new entries produces a new inner node in the tree.

## 7.1 Theoretical Foundations

**Definition 7.1 (Most Specific Common Generalization)** *Let $\Sigma_1$ and $\Sigma_2$ be two substitutions. If there exist substitutions $\mu$, $\sigma_1$, and $\sigma_2$ such that $\sigma_1 \bullet \mu = \Sigma_1$ and $\sigma_2 \bullet \mu = \Sigma_2$ and there is no substitution $\lambda$ such that $\lambda\mu$ has these properties, then*

$$mscg(\Sigma_1, \Sigma_2) := (\mu, \sigma_1, \sigma_2)$$

*The substitution $\mu$ is called the most specific common generalization (mscg). The substitutions $\sigma_1$ and $\sigma_2$ are called specializations.*

The definition is illustrated by the following example: Suppose $\Sigma = \{x \mapsto g(b), y \mapsto a\}$ and $\rho = \{x \mapsto g(a), y \mapsto b\}$. Then $mscg(\Sigma, \rho) := (\{x \mapsto g(x_1)\}, \{x_1 \mapsto b, y \mapsto a\}, \{x_1 \mapsto a, y \mapsto b\})$. The original substitution $\Sigma$ can be reconstructed by $\{x_1 \mapsto b, y \mapsto a\} \bullet \{x \mapsto g(x_1)\}$ and $\rho = \{x_1 \mapsto a, y \mapsto b\} \bullet \{x \mapsto g(x_1)\}$, respectively. Note that $x_1$ is a new auxiliary variable . Obviously, these auxiliary variables represent the parts of the substitutions which differ from each other.

Generally spoken, the insertion process is very similar to finding variant entries in the tree. To this end we define the function $\mathcal{V}$ which tries to match the codomain of the substitutions of a tree to the terms which have to be inserted.

**Definition 7.2 (Variant Nodes)** *Let $N = (\Sigma, \Omega)$ be a substitution tree and $\rho$ a substitution. Then*

$$\mathcal{V}(N, \rho) := \{\, \sigma \mid \forall x_i \in DOM(\Sigma).\ \sigma(\rho(\Sigma(x_i))) = \rho(x_i) \ \wedge\ DOM(\sigma) \cap V^* = \emptyset \}$$

When looking for variant nodes, indicator variables are not bound. However, since the substitution which is inserted has to be normalized before adding it to the substitution tree, the test for variant nodes will succeed in matching two identical indicator variables, because no binding will have to be made.

A heuristic *select_subnode* is used for descending into the tree instead of traversing all possible subnodes of a node in the tree. This heuristic has to cope with three different situations: First of all, the heuristic has to select a variant subnode of the current node for descending if such a variant exists. Second, the heuristic selects a non–variant subnode which will yield a non–empty mscg if a variant couldn't be found. Third, if neither a

variant nor a subnode which yields a non–empty common generalization could be found, the heuristic has to select the empty tree for insertion. In this case our insertion function will create a new leaf node.

The insertion function *ins* returns a tuple $(N, M)$ where $N$ is the modified tree and $M$ is the leaf node which has been found or inserted, respectively. We chose this approach for the following purpose: Usually, people don't just want to store substitutions, but use the index as a means for accessing data. To this end it has to be possible to store additional information at the leaf nodes of the search tree. Therefore, we also return the leaf node $M$ to enable the user to perform some extra operations on this node.

**Definition 7.3 (Insertion Function** *ins***)** *Let $N = (\Sigma, \Omega)$ be a substitution tree, $\rho$ a substitution which is supposed to be inserted and $OV$ the set of open variables at node $N$. Then*

$$ins(N, \rho, OV) := \begin{cases} (M, M) & \\ \qquad \textit{if } N = \varepsilon \textit{ and } M = (\rho|_{OV}, \emptyset) & (1) \\ (N, N) & \\ \qquad \textit{if } \exists \sigma \in \mathcal{V}(\Sigma, \rho) \textit{ and } \Omega = \emptyset & (2) \\ ((\mu, \{(\sigma_1, \Omega), M\}), M) & \\ \qquad \textit{if } \neg\exists \sigma \in \mathcal{V}(\Sigma, \rho) \textit{ and } mscg(\Sigma, \rho) = (\mu, \sigma_1, \sigma_2) & \\ \qquad \textit{and } M = (\sigma_2 \cup \rho|_{OV \setminus DOM(\Sigma)}, \emptyset) & (3) \\ ((\Sigma, \Omega \setminus N' \cup M'), M) & \\ \qquad \textit{if } \exists \sigma \in \mathcal{V}(\Sigma, \rho) \textit{ and } \Omega \neq \emptyset \textit{ and} & \\ \qquad N' = \textit{select\_subnode}(N) \textit{ and} & \\ \qquad (M', M) = ins(N', \sigma\rho, OV \setminus DOM(\Sigma) \cup I(\Sigma)) & (4) \end{cases}$$

To insert a substitution $\rho$ into a tree $(N, M) = ins(N, \overline{\rho}, DOM(\rho))$ is called. Note that the substitution $\rho$ has to be normalized in advance. Rule (1) creates a new leaf node in case a substitution is inserted into an empty tree, while Rule (2) returns a leaf node which corresponds to a variant substitution which has been inserted yet. Rule (3) creates a new inner node and a new leaf in case the substitution in the tree is not a variant of the substitution which has to be inserted. The set of open variables $OV$ is needed for completely describing the inserted substitution in case a new leaf node is created. Finally, Rule (4) uses a heuristic to find a subnode of the current node where the insertion will be continued. Depending on the subnode selected by the heuristic Rule (4) can cause different insertions.

## 7.2 Examples

Consider the exemplifying insertion sequence in Fig. 10. Rule (1) works on empty trees and the insertion of $\{x \mapsto \overline{f(z, g(b))}\}$ into an empty tree yields leaf A marked with $\{x \mapsto f(*_1, g(b))\}$.

The substitution $\{x \mapsto \overline{f(y, g(b))}\}$ is inserted into tree A. The tree is non–empty and $f(*_1, g(b))$ is a variation of the substitution in the root node. In this case no new leaf is created and tree B is identical to tree A.

The substitution $\{x \mapsto \overline{f(a,b)}\}$ is inserted into tree B yielding tree C. As $f(*_1, g(b))$ is not a variant of $f(a,b)$, Rule (3) adds a new root and a new leaf node to the tree. The new root represents the common parts of $f(*_1, g(b))$ and $f(a,b)$. Such a most specific common generalization contains auxiliary variables at the positions where the original terms are different. In our example the root is marked with $\{x \mapsto f(x_1, x_2)\}$. Additionally, the new leaf contains the bindings which are needed to completely represent the substitution being inserted.

Insertion of the substitution $\{x \mapsto \overline{f(c, g(d))}\}$ to tree C employs the heuristic mentioned in Rule (4). As the substitution to be inserted is a variant of the root of the tree, the heuristic has to select the subnode of the tree where the insertion process will be continued. Assume that the node marked with $\{x_1 \mapsto *_1, x_2 \mapsto g(b)\}$ is selected. Again Rule (3) is applied; the resulting tree D contains a new leaf node and a new inner node marked with $\{x_2 \mapsto g(x_3)\}$. This substitution is the most specific common generalization for the substitutions $\{x_1 \mapsto *_1, x_2 \mapsto g(b)\}$ and $\{x_1 \mapsto c, x_2 \mapsto g(d)\}$.

Eventually, $\{x \mapsto \overline{f(b, g(a))}\}$ is inserted into tree D, thus yielding tree E. In this case Rule (4) is applied two times in a row. Assume that the heuristic in the first application selects node $\{x_2 \mapsto g(x_3)\}$ for insertion. However, the heuristic applied to this node cannot find a subnode which is either a variant or at least would produce a non–empty mscg if Rule (3) was applied. Therefore, our heuristic selects the empty tree, and Rule (1) produces a single leaf which then is added to the leaf nodes of the current node by Rule (4).

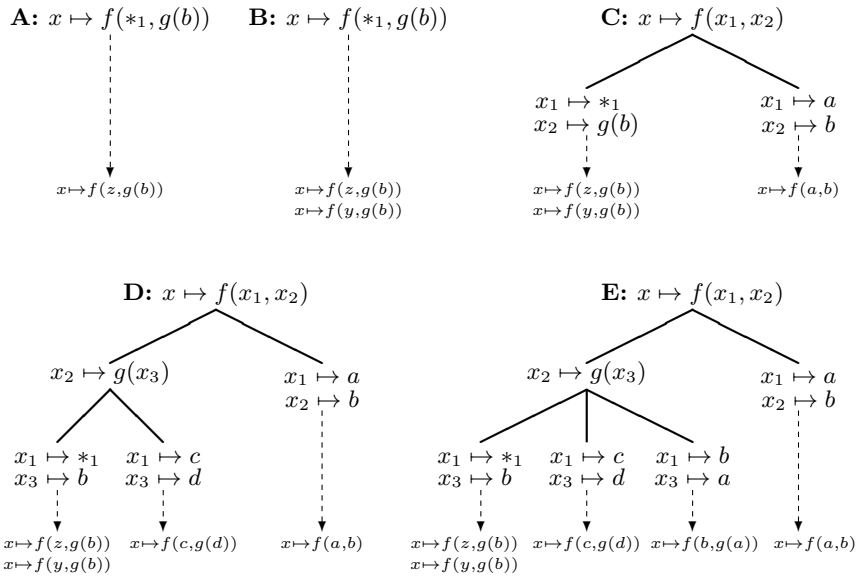



Figure 10: Insertion sequence for substitutions with identical domains

So far, all nodes in our substitution trees have been marked with non–empty substitutions, which is not the case in general! The insertion of the substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$ into an empty tree, for instance, will yield a root node marked with the empty substitution, as the constants $a$ and $b$ actually have very little in common. The same thing happens if substitutions with non–identical domains are inserted into our index. The in-

sertion sequence in Fig. 11 shows more details: First, the substitution $\{x \mapsto g(a), y \mapsto b\}$ is inserted into the empty tree and a single leaf is created. Insertion of $\{x \mapsto g(b), z \mapsto b\}$ produces a root marked with $\{x \mapsto g(x_1)\}$. The substitution $\{z \mapsto a\}$ is neither a variant of $\{x \mapsto g(x_1)\}$ nor is there a non–empty common generalization for the two substitutions; an empty root is created. Finally, $\{x \mapsto g(a)\}$ is added to the tree. However, this substitution is part of a substitution which has been inserted. As a consequence the new leaf node is also marked with an empty substitution. Note that inner nodes never have empty substitutions.

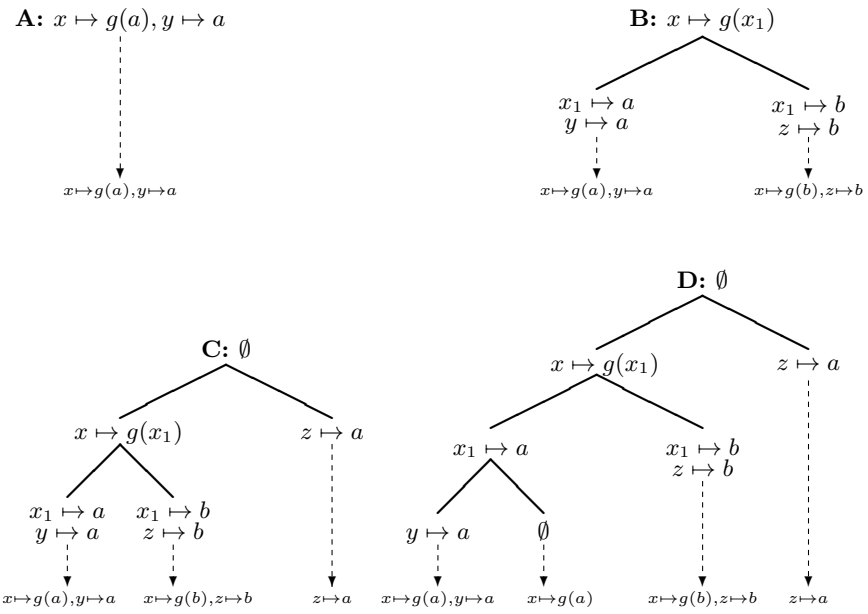



Figure 11: Insertion sequence for substitutions with different domains

## 7.3 Computing the Most Specific Common Generalization

The next two definitions 7.4 and 7.5 are rather technical. They show how the mscg is derived in detail. In Def. 7.4 the function $mscg^t$ which computes the mscg for two terms is defined. Definition 7.5 extends $mscg^t$ to substitutions.

**Definition 7.4 (MSCG for Terms)** *Let $s$ and $t$ be two terms and $\sigma_1$ and $\sigma_2$ be two substitutions. Then*

$$mscg^t(s,t,\sigma_1,\sigma_2) := \begin{cases} (f(cg_1,\ldots,cg_n),\sigma_1,\sigma_2) & \\ \qquad \textit{if } s = f(s_1,\ldots,s_n) \textit{ and } t = f(t_1,\ldots,t_n) \textit{ and} & \\ \qquad \forall_{1\le i\le n}\,(cg_i,\sigma_1,\sigma_2) = mscg^t(s_i,t_i,\sigma_1,\sigma_2) & (1) \\ (s,\sigma_1,\sigma_2) & \\ \qquad \textit{if } s \in V^* \textit{ and } s = t & (2) \\ (s,\sigma_1,\sigma_2 \cup \{s \mapsto t\}) & \\ \qquad \textit{if } s \in V \backslash V^* \textit{ and } s \neq t & (3) \\ (y,\sigma_1,\sigma_2) & \\ \qquad \textit{if } \exists y \in DOM(\sigma_1) \cap DOM(\sigma_2) \textit{ and} & \\ \qquad \sigma_1(y) = s \textit{ and } \sigma_2(y) = t & (4) \\ (x_j,\sigma_1 \cup \{x_j \mapsto s\},\sigma_2 \cup \{x_j \mapsto t\}) & \\ \qquad \textit{where } x_j \textit{ is a new auxiliary variable.} & (5) \end{cases}$$

Rules (1) and (2) extract identical symbols from the two terms with Rule (1) handling constant and function symbols and Rule (2) treating indicator variables as if they were constants. Note that the substitutions $\sigma_1$ and $\sigma_2$ in Rule (1) are changed by every call of $mscg^t$. In case there is a non–indicator variable in term $s$ it can also be used in the specialization for the new leaf node, which is stated by Rule (3). Rule (4) describes how the assignments will be reused in the found specializations, resulting in a non–linear mscg. However, to achieve generalizations this rule may be omitted. The effect on the substitution tree is minimal, because most of the non–linear generalizations in a tree are changed to linear generalizations by further insertions. Changes in memory requirements and retrieval times are minimal. Finally, Rule (5) introduces new non–indicator variables in case none of the other rules could be applied.

When solving $mscg^t(f(a,*_2,x_1,*_1,*_1), f(a,*_2,c,b,b),\emptyset,\emptyset)$, for instance, all five Rules of the definition of $mscg^t$ are applied. The result is $(f(a,*_2,x_1,x_2,x_2),\ \{x_2 \mapsto *_1\},\ \{x_1 \mapsto c,\ x_2 \mapsto b\})$.

**Definition 7.5 (MSCG for Substitutions)** *Let $\Sigma$ be a substitution in a substitution tree, $\rho$ the current variable bindings. The most specific common generalization $\mu$ and the*

*specializations $\sigma_1$ and $\sigma_2$ are computed using the following recursive definition.*

$$mscg^{\sigma}(\Sigma, \rho, \mu, \sigma_1, \sigma_2) := \begin{cases} mscg^{\sigma}(\Sigma \backslash \{x_i \mapsto t_i\}, \rho, \mu, \sigma_1 \cup \{x_i \mapsto t_i\}, \sigma_2) \\ \qquad \textit{if } x_i \mapsto t_i \in \Sigma \textit{ and } x_i = \rho(x_i) & (1) \\ mscg^{\sigma}(\Sigma \backslash \{x_i \mapsto t_i\}, \rho, \mu \cup \{x_i \mapsto t_i\}, \sigma_1, \sigma_2) \\ \qquad \textit{if } x_i \mapsto t_i \in \Sigma \textit{ and } t_i = \rho(x_i) & (2) \\ mscg^{\sigma}(\Sigma \backslash \{x_i \mapsto t_i\}, \rho, \mu, \sigma_1 \cup \{x_i \mapsto t_i\}, \\ \sigma_2 \cup \{x_i \mapsto \rho(x_i)\}) \\ \qquad \textit{if } x_i \mapsto t_i \in \Sigma \textit{ and } top(t_i) \neq top(\rho(x_i)) & (3) \\ mscg^{\sigma}(\Sigma \backslash \{x_i \mapsto t_i\}, \rho, \mu \cup \{x_i \mapsto cg\}, \sigma_1, \sigma_2) \\ \qquad \textit{if } x_i \mapsto t_i \in \Sigma \textit{ and } top(t_i) = top(\rho(x_i)) \\ \qquad \textit{and } t_i \neq \rho(x_i) \textit{ and } (cg, \sigma_1, \sigma_2) = \\ \qquad mscg^{t}(t_i, \rho(x_i), \sigma_1, \sigma_2) & (4) \\ (\mu, \sigma_1, \sigma_2) \\ \qquad \textit{if } \Sigma = \emptyset & (5) \end{cases}$$

The function $mscg^{\sigma}$ considers every assignment $x_i \mapsto t_i \in \Sigma$. We have seen, that during the insertion variable bindings are established by the function $\mathcal{V}$. These current bindings are stored in $\rho$. Rule (1) handles assignments in $\Sigma$ which map variables that don't occur in the substitution which is inserted. Rule (2) detects assignments where the binding of $x_i$ is identical to what it has to be bound. In case the terms under consideration don't even have the same top symbol, Rule (3) completely splits the information to the specializations. Finally, Rule (4) initiates calls to $mscg^{t}$ in all other cases.

For example, we have $mscg^{\sigma}(\{y \mapsto a, x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto f(c)\}, \{x_1 \mapsto a, x_2 \mapsto c, x_3 \mapsto f(a)\}, \emptyset, \emptyset, \emptyset) = (\{x_1 \mapsto a, x_3 \mapsto f(x_4)\}, \{y \mapsto a, x_2 \mapsto b, x_4 \mapsto c\}, \{x_2 \mapsto c, x_4 \mapsto a\})$.

## 7.4 Reusing non–indicator Variables

In Rule (5) of Def. 7.4 we introduced *new auxiliary variables* for representing the differences of the terms in the mscg. However, such a non–indicator variable doesn't really have to be a *new* variable: In tree D in Fig. 10 we introduced the new variable $x_3$. Obviously, the variable $x_3$ could be used again if we had to create another mscg in the right subtree of the root. Generally spoken, let $N_0 = (\Sigma_0, \Omega_0)$, ..., $N_i = (\Sigma_i, \Omega_i)$ be a path in a substitution tree and $N_i$ a node which has to be extended. In this situation the set of non–indicator variables which can be reused is the set of all non–indicator variables in the tree minus the domain variables on the path from $N_0$ to $N_i$ minus the domain variables that occur in the subtree $N_i$. More formally, this set can also be described as

$$DOM(Subst(N_0)) \;\setminus\; \bigcup_{0 \leq j < i} DOM(\Sigma_j) \;\setminus\; DOM(Subst(N_i)).$$

Reusing non–indicator variables is an advantage both for memory requirements and for the speed of the retrieval. Using the above definition the reusable variables can easily be computed during insertion if the set $DOM(Subst(N_0))$ is maintained, which implies that

it doesn't have to be recomputed for each insertion. In our implementation of substitution tree indexing, this technique of variable reusing was able to reduce the number of variables by a factor of 40 in the average case.

## 7.5 Insertion Heuristics

As mentioned, an insertion heuristic is used for descending into the tree and the heuristic has to cope with three different situations: Either it selects a variant subnode of the current node for descending or it selects a non–variant subnode which will yield a non–empty mscg if a variant couldn't be found. If such a node doesn't exist either, the empty tree is selected. In this case the insertion function will create a new leaf node. In our implementation we used a very simple *first–fit* heuristic: We chose the *first*[1] variant son for descending. If such a son doesn't exist the *first* non–variant son which produces a non–empty mscg is selected.

In general, using a more complex insertion heuristic implies the necessity of traversing the tree finding all possible insertion positions. Finally, the different positions found are rated by the insertion heuristic and the substitution is inserted at the best position. However, this technique has some disadvantages compared to the first–fit technique:

- The whole tree has to be traversed.
- The variable bindings found in the traversal either have to be stored for each insertion position or have to be recomputed for the insertion position that was considered to be best.
- Further insertions to the index are not known at the moment the selection of the best node is taken. This implies that the decision to insert the substitution at a specific position could be "wrong".
- Most insertion heuristics cause the computation or the storage of additional information like the depth of a subtree and such.

For all these reasons using insertion heuristics doesn't guarantee best results. Trying several heuristics which minimize

- size of the index,
- number of auxiliary variables $x_i$ in the index,
- number of sons for inner nodes of the tree, or
- depth of indicator variables $*_i$ in the tree

when inserting new substitutions confirmed our suspicion: Insertion was *very* slow and, although the resulting trees were a little smaller with some heuristics, the retrieval times didn't change significantly. Due to this experience, using a complex insertion heuristic cannot be recommended.

[1] In the definition of substitution trees the subtrees of an inner node are represented by a set of subtrees. A real implementation, however, will store these subtrees in some order.

# 8 Deletion

The deletion of entries in a substitution tree is based on two facts. First, the substitution which has to be deleted from the index doesn't need to be normalized if we keep on looking for variants until we find a leaf node in the tree. Second, just like the insertion function, the deletion has to cope with additional information which is stored in the leaf nodes of the tree, if the tree is used to access a database by substitutions. To this end, the deletion uses a predicate $\Delta$ in order to determine whether a specific node really has to be deleted. Assume the user stores different pointers at the leaf nodes of the tree. If a leaf only contains the pointer which was searched, the whole leaf node has to be deleted. If the leaf contains more than this pointer, the pointer has to be deleted from the list of pointers stored at the leaf node, but the node itself is left unchanged.

**Definition 8.1 (Deletion Function** *del***)** *Let $N$ be a substitution tree and $\rho$ a substitution which is supposed to be deleted. Function del deletes nodes which represent variants of $\rho$ if the predicate $\Delta$ evaluates to true at specific leaf nodes in the tree.*

$$
del(N,\rho,\Delta) := \begin{cases}
\varepsilon & \text{if } \exists\sigma \in \mathcal{V}(\Sigma,\rho) \text{ and either } \Omega = \emptyset \text{ and } \Delta(N) \\
 & \text{or } \Omega \neq \emptyset \text{ and } \bigcup_{N_i\in\Omega} del(N_i,\sigma\rho,\Delta) = \emptyset \quad (1)\\
N & \text{if } \exists\sigma \in \mathcal{V}(\Sigma,\rho) \text{ and } \Omega = \emptyset \text{ and } \neg\Delta(N) \quad (2)\\
(\Sigma, \bigcup_{N_i\in\Omega} del(N_i,\sigma\rho,\Delta)) & \\
 & \text{if } \exists\sigma \in \mathcal{V}(\Sigma,\rho) \text{ and } \Omega \neq \emptyset \\
 & \text{and } |\bigcup_{N_i\in\Omega} del(N_i,\sigma\rho,\Delta)| \geq 2 \quad (3)\\
(\Sigma' \bullet \Sigma, \Omega') & \\
 & \text{if } \exists\sigma \in \mathcal{V}(\Sigma,\rho) \text{ and } \Omega \neq \emptyset \\
 & \text{and } \bigcup_{N_i\in\Omega} del(N_i,\sigma\rho,\Delta) = \{(\Sigma',\Omega')\} \quad (4)\\
N & \text{otherwise} \quad (5)
\end{cases}
$$

Rule (1) is applied either if $N$ is a leaf node and $\Delta$ decides to delete the leaf or if all subtrees of an inner node are deleted by recursive calls of *del*. Leaf nodes where $\Delta(N)$ evaluates[2] to *false* are left unchanged, as stated by Rule (2). Rule (3) describes the deletion of subnodes of an inner node where at least two subtrees remain. The join in Rule (4) is needed in case just one subtree is left over, because our definition of substitution trees demands that every inner node has to have at least two subtrees. In case the current node $N$ is not a variant of the current variable bindings $\rho$ the node is left unchanged, which is stated by Rule (5). Obviously, Fig. 10 and Fig. 11 also represent deletion sequences if the trees are read in reverse order.

**Deleting the Results of a Retrieval.** The function *del* can easily be modified so it will delete all instances, generalizations, or unifiable entries from the index. Simply change all occurrences of $\mathcal{V}(\Sigma,\rho)$ to $\mathcal{I}(\Sigma,\rho)$, $\mathcal{G}(\Sigma,\rho)$, or $\mathcal{U}(\Sigma,\rho)$, respectively. Additionally, $\Delta(N)$ has to be *true* for all nodes $N$.

[2]Maybe after having performed some side effects on the data stored at the leaf.

# 9 Experiments

In this section substitution trees are compared to other indexing methods. As most of the other techniques don't serve as an indexing mechanism for substitutions, we use our implementation of substitution trees as a term index.

## 9.1 The Term Sets

For the experiments special term sets were used. Part of them have been introduced in [McC92]. These sets were taken from typical OTTER applications. As the sets are paired, there is a set of positive literals and a set of negative literals in each pair. Unifiable terms are searched in order to find resolution partners and to detect unit conflicts. The sets EC–pos and EC–neg consist of 500 terms each and are derived from a theorem in equivalential calculus. Two representative members of EC–pos and EC–neg are

$$P(e(e(x,e(y,e(z,e(e(u,e(v,z)),e(v,u))))),e(y,x)))$$

and

$$\neg P(e(e(x,e(e(y,e(z,x)),e(z,y))),e(e(u,e(e(v,e(w,u)),e(w,v))),e(e(v6,\\ e(e(v7,e(v8,v6)),e(v8,v7))),e(e(v9,e(e(e(b,a),\\ e(e(e(a,e(b,c)),c),v9)),v10)),v10))))).$$

The sets CL–pos and CL–neg have 1000 members and are derived from a theorem in combinatory logic. Two representative members of CL–pos and CL–neg are

$$g(x,g(g(g(g(g(B,B),y),z),u),v)) = g(g(g(B,x),g(y,z)),g(u,v)$$

and

$$g(f(g(g(N,x),y)),g(g(g(N,x),y),f(g(g(N,x),y)))) \neq \\ g(g(g(x,f(g(g(N,x),y))),y),f(g(g(N,x),y))).$$

The sets BOOL–pos and BOOL–neg are derived from a theorem in the relational formulation of Boolean algebra and consist of 6000 terms each. Two representative members of BOOL–pos and BOOL–neg are

$$Sum(x,p(x,y),p(x,s(x,y)))$$

and

$$\neg Sum(p(c2,n(x)),p(c2,x),c4).$$

The other part of the term sets was produced randomly. All of these sets contain 10000 terms. Three different function symbols with varying arities and three different constants have been used. The terms contain at most three different variables with possibly multiple occurrences.

The set WIDE-10000 contains function symbols with an arity of at least 3. In the other sets the maximal arity of function symbols is 2.

The maximal depth of all terms is 3 except for the set DEEP-10000 where the maximal term depth is 6.

Terms in LIN–10000 contain each variable at most once and the set GND-10000 contains no variables at all. All other sets contain linear as well as non–linear terms.

For example, a representative member of the set AVG-10000 which contains linear and non–linear terms with a maximal depth of 3 and a maximal arity of 2 is

$$g(f(g(x14, a), g(a, c)), g(h(x14), f(b, c))).$$

## 9.2 Memory Requirements

In Fig. 12 the memory requirements of discrimination trees (DT), abstraction trees (AT), and substitution trees (ST) are compared. Additionally, a survey on the times needed to build the index and the times needed to delete all entries of the index one by one is presented.

**Memory Requirements [KBytes]**

| **Indexed Set** | DT | AT | ST |
|---|---|---|---|
| EC–pos | 108 | 153 | **88** |
| EC–neg | 539 | 483 | **252** |
| CL–pos | 612 | 505 | **318** |
| CL–neg | 2113 | 1116 | **797** |
| BOOL–pos | 685 | 780 | **610** |
| BOOL–neg | 958 | 1090 | **858** |
| AVG–10000 | 969 | 1278 | **935** |
| WIDE–10000 | 13402 | 9312 | **8056** |
| GND–10000 | 846 | 1015 | **831** |
| LIN–10000 | 1020 | 1265 | **950** |
| DEEP–10000 | 6191 | 4312 | **3766** |

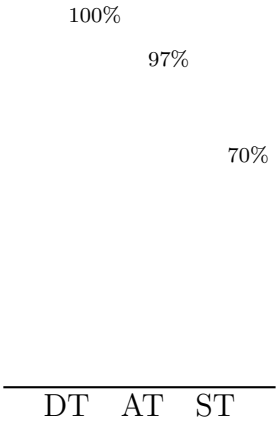



Figure 12: Memory requirements

First of all, our experiments show that substitution trees require the least memory space. This result was expected, because the information sharing is best in this index. In the average case a substitution tree consumes 70% of the memory occupied by the worst of the three techniques. This fact is illustrated in Fig. 12. For each of the three indexing techniques we have three bars. The white bar represents the average behavior, for example discrimination tree indexing is most greedy in memory consumption. The gray bar shows the best result of the experiments and the black bar the worst behavior. Consider the bars for abstraction tree indexing: In the average case, the trees need 97% of the memory needed by the most greedy technique. However, there was an experiment where abstraction trees occupied just 53% of the memory occupied by the worst technique. Nevertheless, the black bar tells us that abstraction trees also have at least once been the most memory consuming index.

## 9.3 Insertion

The creation of discrimination trees is fastest, because these indexes are deterministic. To insert a term we just have to normalize it and to insert it at the only possible position in the index. The other two indexing techniques are slower at adding entries to the index, because these indexes are non-deterministic and a position for insertion has to be found using a more or less complex algorithm. In contrast to substitution trees the abstraction tree technique doesn't require a normalization of the entries to be inserted, and therefore abstraction trees are faster at inserting new entries as shown in Fig. 13.

**Insertion [Seconds]**

| **Indexed Set** | DT | AT | ST |
|---|---|---|---|
| EC–pos | **0.3** | **0.3** | **0.3** |
| EC–neg | 1.0 | **0.9** | 1.3 |
| CL–pos | **0.8** | 0.9 | 1.3 |
| CL–neg | **2.4** | **2.4** | 3.2 |
| BOOL–pos | **1.1** | 1.6 | 2.4 |
| BOOL–neg | **1.7** | 2.3 | 3.6 |
| AVG–10000 | **1.7** | 6.4 | 4.2 |
| WIDE–10000 | **9.0** | 13.6 | 17.7 |
| GND–10000 | **1.5** | 2.6 | 4.0 |
| LIN–10000 | **1.8** | 6.0 | 4.3 |
| DEEP–10000 | **5.3** | 11.3 | 9.3 |

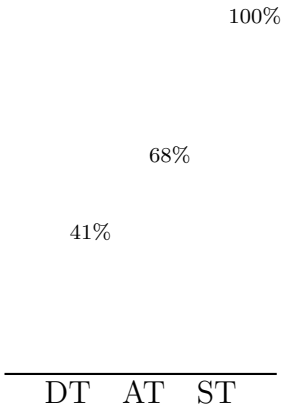



Figure 13: Experiments with insertion

## 9.4 Deletion

Figure 14 shows, that substitution trees are appropriate also for dynamic data, because entries are deleted most quickly although insertion is slow.

**Deletion [Seconds]**

| **Indexed Set** | DT | AT | ST |
|---|---|---|---|
| EC–pos | 0.4 | 0.2 | **0.1** |
| EC–neg | 1.6 | 1.2 | **0.2** |
| CL–pos | 1.6 | 1.0 | **0.1** |
| CL–neg | 4.1 | 3.1 | **1.5** |
| BOOL–pos | **1.9** | 2.2 | 2.7 |
| BOOL–neg | 2.7 | **2.1** | 3.0 |
| AVG–10000 | **2.7** | 6.6 | 4.7 |
| WIDE–10000 | 19.2 | 16.9 | **13.3** |
| GND–10000 | **2.4** | **2.4** | 4.0 |
| LIN–10000 | **2.7** | 6.2 | 4.6 |
| DEEP–10000 | 9.9 | 11.0 | **7.6** |

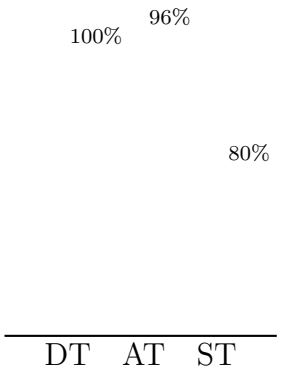



Figure 14: Experiments with deletion

The wide range of the values for deletion in substitution trees is striking; the best

deletion takes just 6% of the time of the worst while there are examples where the deletion in substitution trees itself is slowest. When no indicator variables occur in the terms, as in GND–10000, deletion of entries is slow, because the treatment of indicator variables is the main advantage of substitution tree indexing. Deletion in substitution trees performs extremely well in case the terms are deep as in EC–neg, CL–pos, and DEEP–10000.

## 9.5 Retrieval Times

The experiments were run on a Sun SPARCstation SLC computer with 16 MBytes of RAM. In all but the merge experiments the set in column *Index* was stored in an index. Then for all members of the set *Query* we were looking for more general substitutions, instances, and compatible substitutions. Similar tests using discrimination tree indexing and path–indexing have been reported in [McC92].

**Generalizations [Seconds]**

| Index | Query | DT | AT | ST |
|---|---|---|---|---|
| EC–pos | EC–pos | 0.6 | 0.4 | **0.3** |
| EC–pos | EC–neg | 1.7 | 0.8 | **0.4** |
| EC–neg | EC–pos | 0.4 | **0.1** | **0.1** |
| EC–neg | EC–neg | 2.1 | 0.7 | **0.6** |
| CL–pos | CL–pos | 3.4 | **0.8** | **0.8** |
| CL–pos | CL–neg | 4.0 | **0.2** | 0.3 |
| CL–neg | CL–pos | 1.3 | **0.1** | **0.1** |
| CL–neg | CL–neg | 5.6 | 1.7 | **1.5** |
| BOOL–pos | BOOL–pos | **3.3** | 3.5 | 3.4 |
| BOOL–pos | BOOL–neg | 3.6 | 3.1 | **3.0** |
| BOOL–neg | BOOL–pos | 1.7 | 0.9 | **0.8** |
| BOOL–neg | BOOL–neg | 3.6 | 2.3 | **2.1** |
| AVG–10000 | AVG–10000 | 13.5 | 28.0 | **13.3** |
| WIDE-10000 | WIDE-10000 | 27.9 | **16.2** | 18.0 |
| GND–10000 | GND–10000 | 4.6 | 4.1 | **3.7** |
| LIN–10000 | LIN–10000 | 13.0 | 25.5 | **12.8** |
| DEEP–10000 | DEEP–10000 | 22.7 | 28.2 | **14.8** |

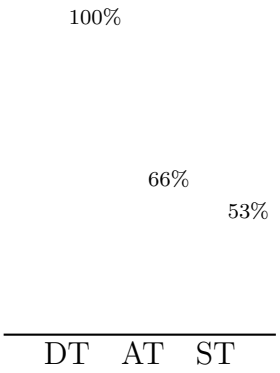



Figure 15: Retrieval of more general terms

By the way, in all experiments path–indexing was much slower than substitution tree indexing. For the merge experiments an index for the set *Index* and an index for the *Query* set was created. Then the two indexes were merged. The retrieval times exclude the time for the creation of indexes. First of all, in the average case substitution tree indexing is the fastest of the techniques. Discrimination trees are slowest. Abstraction trees seem to work well on "wide" terms. The merge of discrimination trees is not defined.

There are only three experiments in which substitution tree indexing is not the fastest technique for the retrieval of more general entries. Due to the introduction of indicator variables, the search for instances using substitution trees takes just 1% of the time of discrimination or abstraction trees in case of the sets EC–neg and EC–pos which contain lots of variables. Substitution trees provide fastest retrieval of instances in all examples.

There are only two experiments where substitution trees don't find unifiable entries most quickly. In eight experiments, the retrieval time using abstraction trees for the merge

**Instances [Seconds]**

| Index | Query | DT | AT | ST |
|---|---|---|---|---|
| EC–pos | EC–pos | 6.5 | 6.0 | **1.0** |
| EC–pos | EC–neg | 3.8 | 6.0 | **0.8** |
| EC–neg | EC–pos | 42.8 | 14.9 | **0.3** |
| EC–neg | EC–neg | 76.4 | 18.6 | **1.4** |
| CL–pos | CL–pos | 17.9 | 4.6 | **1.7** |
| CL–pos | CL–neg | 3.2 | 3.5 | **0.1** |
| CL–neg | CL–pos | 100.0 | 11.9 | **6.7** |
| CL–neg | CL–neg | 5.2 | 5.1 | **3.1** |
| BOOL–pos | BOOL–pos | 6.1 | 7.5 | **5.2** |
| BOOL–pos | BOOL–neg | 2.6 | 4.2 | **2.3** |
| BOOL–neg | BOOL–pos | 18.7 | 2.4 | **1.8** |
| BOOL–neg | BOOL–neg | 3.7 | 3.5 | **2.6** |
| AVG–10000 | AVG–10000 | 60.8 | 50.6 | **32.2** |
| WIDE-10000 | WIDE-10000 | 550.3 | 60.0 | **46.7** |
| GND–10000 | GND–10000 | 5.0 | 5.5 | **4.3** |
| LIN–10000 | LIN–10000 | 49.0 | 44.3 | **30.0** |
| DEEP–10000 | DEEP–10000 | 643.1 | 70.4 | **52.8** |

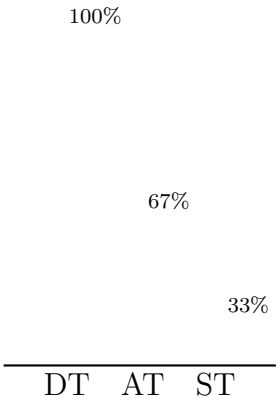



Figure 16: Retrieval of instances

**Unifiable Terms [Seconds]**

| Index | Query | DT | AT | ST |
|---|---|---|---|---|
| EC–pos | EC–pos | 27.9 | 32.1 | **11.8** |
| EC–pos | EC–neg | 28.6 | 60.6 | **21.8** |
| EC–neg | EC–pos | 99.9 | 89.2 | **5.8** |
| EC–neg | EC–neg | 308.8 | 211.1 | **46.1** |
| CL–pos | CL–pos | 50.2 | 12.7 | **6.8** |
| CL–pos | CL–neg | 42.7 | 17.6 | **4.8** |
| CL–neg | CL–pos | 309.4 | 22.5 | **16.9** |
| CL–neg | CL–neg | 19.1 | 7.3 | **6.8** |
| BOOL–pos | BOOL–pos | 11.1 | 10.0 | **9.2** |
| BOOL–pos | BOOL–neg | **4.9** | 5.5 | 5.3 |
| BOOL–neg | BOOL–pos | 23.9 | 2.4 | **2.3** |
| BOOL–neg | BOOL–neg | 4.0 | 3.5 | **3.3** |
| AVG–10000 | AVG–10000 | 100.2 | 86.5 | **61.5** |
| WIDE-10000 | WIDE-10000 | 672.7 | **84.3** | 110.3 |
| GND–10000 | GND–10000 | 4.6 | 5.1 | **4.5** |
| LIN–10000 | LIN–10000 | 74.7 | 71.1 | **52.2** |
| DEEP–10000 | DEEP–10000 | 736.6 | 142.3 | **138.6** |


Figure 17: Retrieval of unifiable terms

| Merge [Seconds] | | | |
|---|---|---|---|
| Index | Query | AT | ST |
| EC–pos | EC–pos | 34.9 | **12.1** |
| EC–pos | EC–neg | 89.1 | **8.9** |
| EC–neg | EC–pos | 88.5 | **6.1** |
| EC–neg | EC–neg | 226.3 | **25.9** |
| CL–pos | CL–pos | 11.7 | **4.3** |
| CL–pos | CL–neg | 19.7 | **9.5** |
| CL–neg | CL–pos | 19.2 | **8.3** |
| CL–neg | CL–neg | 5.8 | **4.4** |
| BOOL–pos | BOOL–pos | 9.7 | **8.7** |
| BOOL–pos | BOOL–neg | **3.0** | 3.9 |
| BOOL–neg | BOOL–pos | **3.7** | 5.0 |
| BOOL–neg | BOOL–neg | 1.7 | **1.6** |
| AVG–10000 | AVG–10000 | 93.6 | **57.9** |
| WIDE-10000 | WIDE-10000 | **173.4** | 290.0 |
| GND–10000 | GND–10000 | **2.8** | 3.8 |
| LIN–10000 | LIN–10000 | 76.3 | **48.0** |
| DEEP–10000 | DEEP–10000 | 184.9 | **99.7** |

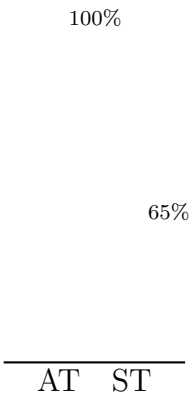



Figure 18: Retrieval of unifiable terms using the merge operation

is slower as the standard retrieval for compatible substitutions. With substitution trees this happened just four times.

### 9.6 Implementation

Substitution trees, abstraction trees, discrimination trees, and (extended) path–indexing are implemented in C and are available via anonymous ftp. They are as well as implementations of other indexing techniques part of "A Collection of Indexing Data Structures (ACID)" developed at MPI. Some of the techniques support subterm retrieval. In the future ACID which is a library for efficient data structures and algorithms for theorem provers will be further improved. Our implementations do not depend on term data structures and can very easily be embedded into other software. For more information send e-mail to *acid@mpi-sb.mpg.de*.

## 10 Conclusion

The new data structure of substitution trees for indexing substitutions was presented. Substitution trees are based on a simple data structure. Experiments showed, that substitution trees perform very well on completely different tasks. They are stable also for large sets of entries and memory requirements are low. Additionally, retrieval is fastest. The disadvantage of relatively slow insertion is compensated by a very fast and powerful deletion procedure.

## References

[BCR93] L. Bachmair, T. Chen, and I.V. Ramakrishnan. Associative-commutative discrimination nets. In *Proceedings TAPSOFT '93, LNCS 668*, pages 61–74. Springer Verlag, 1993.

[BO94] R. Butler and R. Overbeek. Formula databases for high–performance resolution/paramodulation systems. *Journal of Automated Reasoning*, 12:139–156, 1994.

[Chr93] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, February 1993.

[Gra92] P. Graf. Path indexing for term retrieval. Technical Report MPI-I-92-237, Max-Planck-Institut für Informatik, Saarbrücken, Germany, December 1992.

[Gra94] P. Graf. Extended path–indexing. In *12th Conference on Automated Deduction*, pages 514–528. Springer LNAI 814, 1994.

[LO80] E. Lusk and R. Overbeek. Data structures and control architectures for the implementation of theorem proving programs. In *5th International Conference on Automated Deduction*, pages 232–249. Springer Verlag, 1980.

[McC92] W. McCune. Experiments with discrimination-tree indexing and path-indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992.

[Ohl90a] 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.

[Ohl90b] H.J. Ohlbach. Compilation of recursive two-literal clauses into unification algorithms. In *Proc. of AIMSA 1990*. Bulgaria, 1990.

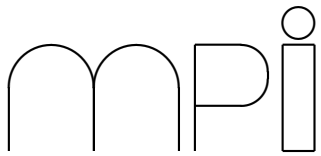
[Rob65] J.A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Sti89] M. Stickel. The path–indexing method for indexing terms. Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.

[Vor94] A. Voronkov. The anatomy of vampire: Implementing bottom–up procedures with code trees. *Submitted to Journal of Automated Reasoning*, 1994.

[WOL91] L. Wos, R. Overbeek, and E. Lusk. Subsumption, a sometimes undervalued procedure. In J.L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 3–40. MIT Press, Cambridge, Massachusetts, 1991.

# Index




Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Regina Kraemer
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `kraemer@mpi-sb.mpg.de`

---

| | | |
|---|---|---|
| MPI-I-94-246 | M. Hanus | On Extra Variables in (Equational) Logic Programming |
| MPI-I-94-241 | J. Hopf | Genetic Algorithms within the Framework of Evolutionary Computation: Proceedings of the KI-94 Workshop |
| MPI-I-94-240 | P. Madden | Recursive Program Optimization Through Inductive Synthesis Proof Transformation |
| MPI-I-94-239 | P. Madden, I. Green | A General Technique for Automatically Optimizing Programs Through the Use of Proof Plans |
| MPI-I-94-238 | P. Madden | Formal Methods for Automated Program Improvement |
| MPI-I-94-235 | D. A. Plaisted | Ordered Semantic Hyper-Linking |
| MPI-I-94-234 | S. Matthews, A. K. Simpson | Reflection using the derivability conditions |
| MPI-I-94-233 | D. A. Plaisted | The Search Efficiency of Theorem Proving Strategies: An Analytical Comparison |
| MPI-I-94-232 | D. A. Plaisted | An Abstract Program Generation Logic |
| MPI-I-94-230 | H. J. Ohlbach | Temporal Logic: Proceedings of the ICTL Workshop |
| MPI-I-94-229 | Y. Dimopoulos | Classical Methods in Nonmonotonic Reasoning |
| MPI-I-94-228 | H. J. Ohlbach | Computer Support for the Development and Investigation of Logics |
| MPI-I-94-226 | H. J. Ohlbach, D. Gabbay, D. Plaisted | Killer Transformations |
| MPI-I-94-225 | H. J. Ohlbach | Synthesizing Semantics for Extensions of Propositional Logic |
| MPI-I-94-224 | H. Aït-Kaci, M. Hanus, J. J. M. Navarro | Integration of Declarative Paradigms: Proceedings of the ICLP'94 Post-Conference Workshop Santa Margherita Ligure, Italy |
| MPI-I-94-223 | D. M. Gabbay | LDS – Labelled Deductive Systems: Volume 1 — Foundations |
| MPI-I-94-218 | D. A. Basin | Logic Frameworks for Logic Programs |
| MPI-I-94-216 | P. Barth | Linear 0-1 Inequalities and Extended Clauses |
| MPI-I-94-209 | D. A. Basin, T. Walsh | Termination Orderings for Rippling |