MAX-PLANCK-INSTITUT FÜR INFORMATIK

Path Indexing for Term Retrieval

Peter Graf

MPI–I–92–237 December 1992

Im Stadtwald
W 6600 Saarbrücken
Germany

Author's Address

Peter Graf (graf@mpi-sb.mpg.de)
Max–Planck–Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken 11
Germany

Publication Notes

This report is a version of chapter 6 of my master's degree thesis "Unification Using Dynamic Sorts" in computer science prepared at the University of Kaiserslautern in June 1992.

Acknowledgements

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

Abstract

Different methods for term retrieval in deduction systems have been introduced in literature. This report reviews the three indexing techniques discrimination indexing, path indexing, and abstraction tree indexing. A formal approach to path indexing is presented and algorithms as well as data structures of an existing implementation are discussed. Eventually, experiments will show that our implementation of path indexing outperforms the implementation of path indexing in the OTTER theorem prover.

Keywords

Term Retrieval, Discrimination Tree Indexing, Path Indexing, Abstraction Tree Indexing, Query Tree.

1 Introduction

Users interact with deduction systems by evaluating failures and trying alternative searches with different strategies or axiom sets. In order to be acceptable as an interacting system the most important feature of a deduction system therefore is execution speed. One can increase the performance of a theorem prover by speeding up the retrieval and maintenance of data referred to by the deduction system. A tool which provides such a fast access to special entries of a database is called *index*.

Indexing may be exploited in different tasks in automated reasoning. In order to find resolution partners for a given literal, for example, a theorem prover has to search for literals with a high unification probability. Subsumption of clauses can be detected by the retrieval of generalizations (forward subsumption) or instances (backward subsumption) of literals of clauses. The retrieval of rewrite rules, demodulators, or, when dealing with sorted logic, the retrieval of term declarations can be accelerated by indexing as well.

We will now explain the different queries which can be started. Although any data may be indexed, we assume ordinary terms to be the entries of our index, which will ease understanding. A query to an index is determined by a tuple $(s, mode)$ with s being a *query term* and *mode* being one of the modes *UNIFIES*, *GENERALIZATION*, and *INSTANCE*. Depending on the mode we result in three different retrievals:

- Find entries t such that t *UNIFIES* with s.
- Find entries t such that t is a *GENERALIZATION* of s.
- Find entries t such that t is an *INSTANCE* of s.

Different methods for term retrieval in deduction systems have been introduced in literature. Some of these are:

- Discrimination Tree Indexing
- Abstraction Tree Indexing
- Path Indexing
- Coordinate Indexing (see [Sti89])
- Codeword Indexing (see [Sti89])

In the following chapter the first three approaches will be discussed.

1.1 Foundations

Before we get into the details of indexing, we want to give a short definition of the notions used:

A *term* is a variable, a constant, or a complex term which consists of a fixed–arity function symbol and a sequence of terms. *Variables* are distinguished from constants by starting with a lower–case letter from u to z. Two terms are *variants* of each other if they are equal modulo variable renaming. Two terms s and t *unify* if there is a substitution σ such that $\sigma(s) = \sigma(t)$. In this context two types of failures arise: On one hand, we have the *occur check* failure which is detected by considering partial substitutions (For example, the unification of $f(x, x)$ and $f(y, g(y))$ will fail since y occurs in $g(y)$.). On the other hand, *clashes* occur when two non–variable symbols cannot be unified because they are distinct. Indirect clashes are again detected by considering partial substitutions (Compare the two terms $f(x, x)$ and $f(a, b)$). A term t is a *generalization* of a term s if there is a substitution σ such that $\sigma(t) = s$. The term s is then called an *instance* of t.

2 Survey on Indexing Techniques

2.1 Discrimination Tree Indexing

In this approach the index is a single tree representing the structure of all the terms which are stored using the index. Every leaf contains a pointer to a set of entries. In the discrimination tree all variables are replaced by the special symbol $*$. Therefore the values of variables are ignored during retrieval and each path from the root of the tree to a leaf corresponds to a set of terms which have variables at the same positions and are syntactically equal in non–variable positions. In figure 1 a discrimination tree is depicted. To answer a query we have to traverse the tree, finding

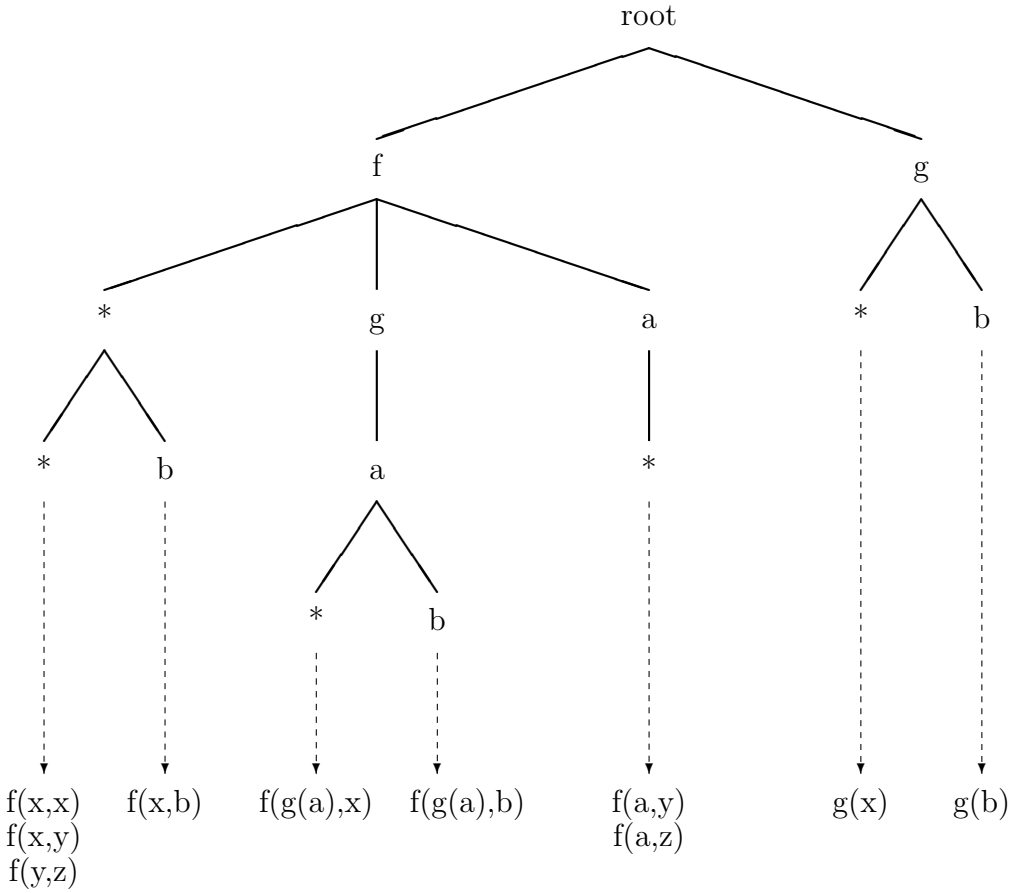

Figure 1: Discrimination Tree

appropriate terms. Let us take the retrieval of generalizations of a query term for an example. First we transform the query term into a structure which is compatible with the discrimination tree. Such a *flat term* is created by simply inserting the query term into an empty discrimination tree. We start the retrieval with 2 pointers to the roots of both the flat term and the discrimination tree. A backtracking algorithm is used to traverse the tree. In this context we have to consider that a subterm of the query term can match $*$–nodes and non–variable nodes in the discrimination tree. In this case both branches must be explored. A problem arises as we have to find the end of subterms in the discrimination tree. For that purpose one could compute the end of the subterm, because we have to deal with fixed–arity function symbols. Another possibility is to maintain a *jump list* for each node of the discrimination tree which contains a pointer to the end of every subterm which starts at this node.

Note that the retrieval is not always correct, because variables are substituted by $*$. Therefore, the index is used as a filter which returns candidates only. Occurs check failures and indirect clashes still have to be detected.

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. As in generalization retrieval, the end of a subterm has to be found when a variable is matched. However, this time the variable occurs in the query term!

The retrieval of unifiable terms searches for generalizations as well as for instances of subterms. Again, we have to jump over segments of the index tree, perform an occur check and call a regular unification routine.

The memory requirement depends on the sharing of initial "substrings" of the target terms stored in the index. Discrimination indexing is faster [McC92] than path indexing in finding generalizations of a query term. More details about discrimination tree indexing can be found in [LO80].

2.2 Abstraction Tree Indexing

Abstraction tree indexing [Ohl90] exploits the lattice structure of terms. It is implemented with the help of a data structure called *abstraction tree*. The abstraction tree is based on the usual instance relation which forms a partial ordering. For example, the three terms $f(h(x), b)$, $f(h(a), b)$, and $f(h(b), b)$ can be ordered according to their instance relation. Obviously, we have $f(h(x), b) > f(h(a), b)$ and $f(h(x), b) > f(h(b), b)$. This ordering can easily be represented by a tree with the root $f(h(x), b)$ and the leaves $f(h(a), b)$ and $f(h(b), b)$. But this structure is not yet an abstraction tree. Actually, the tree contains redundant information, as we know that the leaves are instances of the root. Therefore, we do not store $f(h(a), b)$ and $f(h(b), b)$ in the abstraction tree, but the substitutions on the variable x in $f(h(x), b)$ such that the leaves can be reproduced by the root and the according substitution. In general an abstraction tree is a tree whose nodes are labelled with termlists such that the free variables of the termlists at node N and the termlists of N's subnodes form the domain and codomain of a substitution or more precisely a matcher. An abstraction tree is depicted in figure 2. Note that auxiliary variables are written in italic style.

The procedure for accessing unifiable terms takes a node N and a termlist. It unifies[1] the termlist with N's label. One unifier after the other is applied 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.

Finding instances is also similar to finding unifiable terms. The only difference is that we use matching instead of unification at the leaf nodes.

We have to pay a high price for the comfort in term retrieval: Insertion and deletion are more expensive as in most other techniques (e.g. path indexing). Imagine to search the node N in the abstraction tree which represents the most specific generalization of the term to be inserted. Some nondeterminism remains, e.g. there can be more than one most specific generalization for a term in the abstraction tree. Although insertion is a complex task, we expect the abstraction tree indexing technique to perform well compared to path indexing and discrimination tree indexing for special applications. For example, maintaining the literals of clauses in the abstraction tree offers the possibility of finding generalizations or instances of the term which has to be inserted at insertion time. We see: No extra subsumption test has to be started. Additionally, retrieval for a single query term can be extended to the merging of two abstraction trees b_1 and b_2, which corresponds to the retrieval of partners in b_1 for all terms stored in b_2. This operation might be interesting for finding resolution partners for all literals of a clause in parallel.

2.3 Path Indexing

The index consists of a list of secondary indices, so–called *path lists*. A path list is a sorted list of pointers to terms that share a special path property (i.e. the second argument of a term is the constant a). Path properties are written as *keys* (e.g. [g], [g1*], ...).

[1] Abstraction tree indexing works for every finitary unification theory.

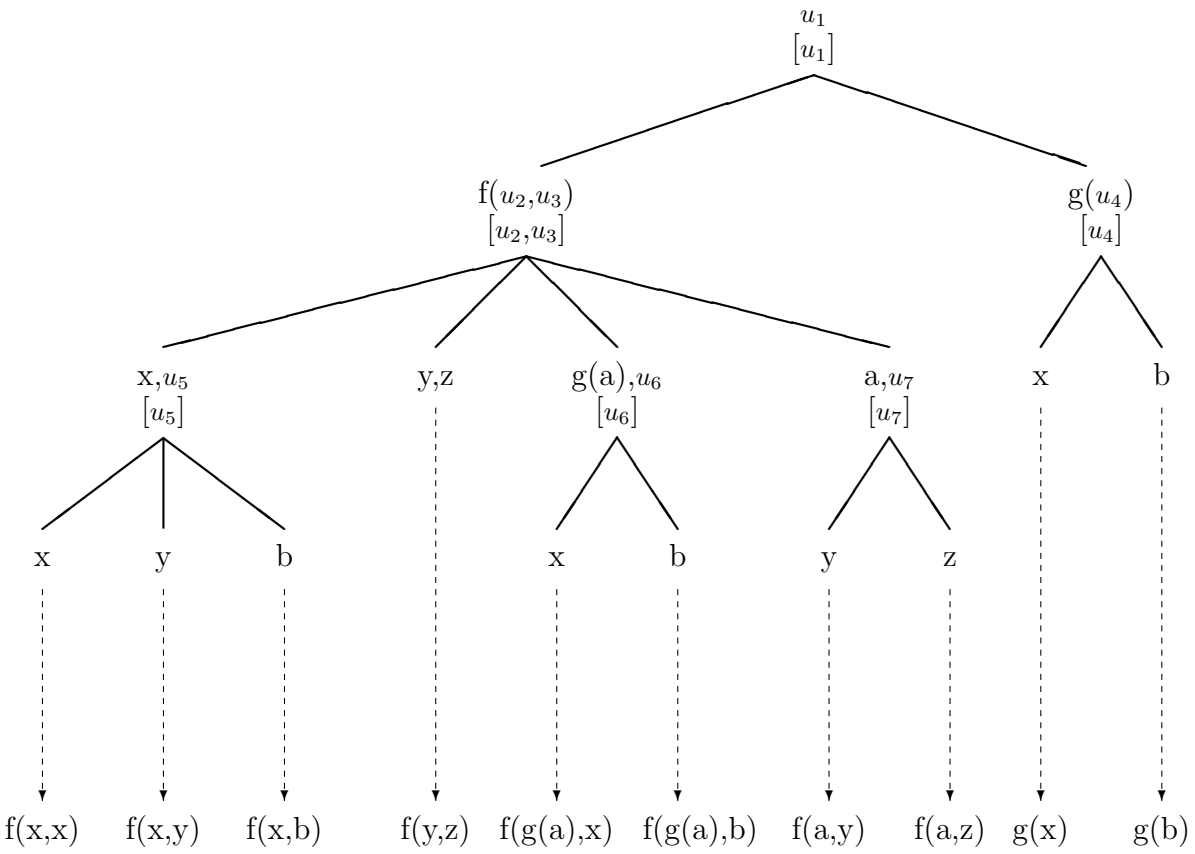

Figure 2: Abstraction Tree

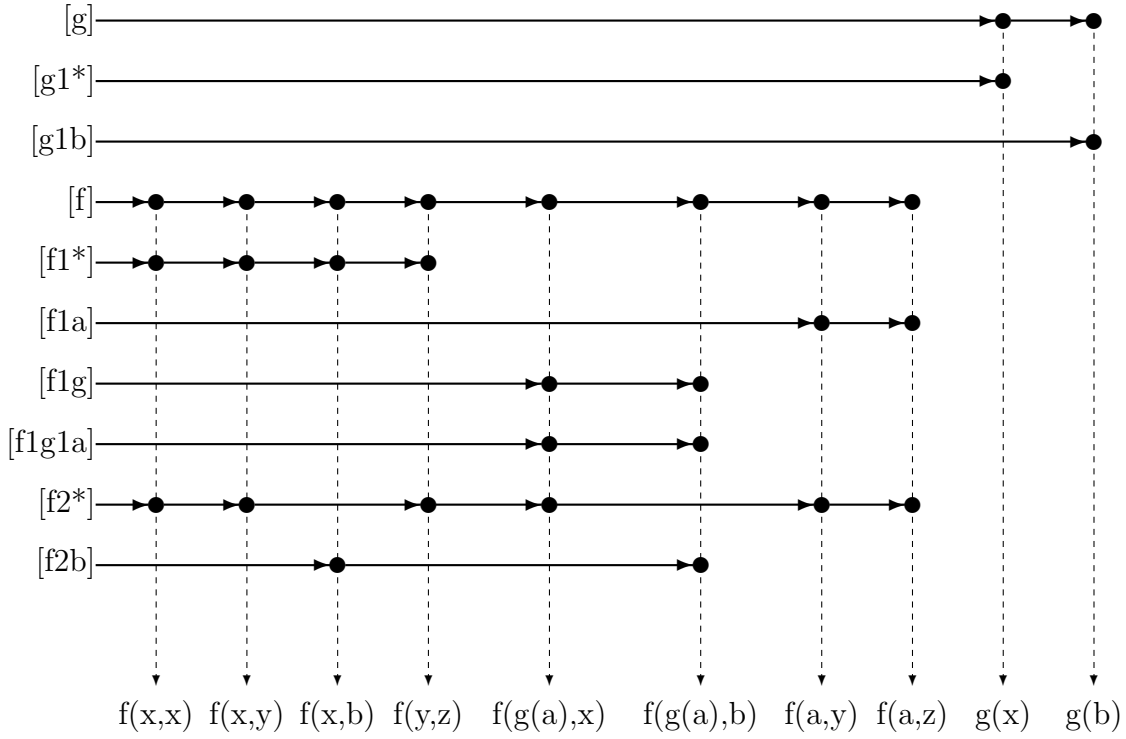

Figure 3: Path Index

To find a specific path list, we use a hashing mechanism. We index terms with special path properties by following the pointers of each path list. These pointers lead to terms with the path properties we are looking for. If a term is pointed to by every path list we take into consideration, it has all the desired path properties. We will have to compute unions and intersections of path lists in order to find terms that match or unify with a given term or are generalizations of a term, respectively. The path index is not a perfect filter, terms found in the index are candidates for the required purpose only. Besides, the occur check will have to be performed and indirect clashes will have to be detected. Insertion of entries is easy. We simply add pointers to all path lists which lead to entries having the path property of the entry to be inserted. Deletion can be performed as quickly. Figure 3 shows a path index.

A variation on path indexing is the possibility of limiting the indexing depth. For example depth 0 considers the outermost function symbol only, depth 1 works on the outermost function symbol and the immediate subterms, and so on. This technique decreases memory requirements but increases the number of incorrect retrievals.

2.4 Comparison

Eventually, we present a comparison between the three approaches to term indexing. On one hand, we have seen that insertion and deletion of entries is more complex in abstraction tree indexing than in path indexing or discrimination tree indexing. On the other hand, abstraction tree indexing provides forward and backward subsumption tests at insertion time. Discrimination tree indexing is usually faster than path indexing in finding generalizations of a query term [McC92]. Abstraction tree indexing is a perfect[2] filter, because full unification is applied at the nodes of the tree. Path and discrimination tree indexing are not perfect filters. Since path indexing offers the possibility of limiting the index depth, we can choose between little storage requirements and a low rate of incorrect retrievals. Probably, abstraction tree indexing results in the lowest storage requirements of the three approaches. An advantage of abstraction tree indexing is the ability of retrieving partners for a set of query terms by the merging of two abstraction trees. Additionally, abstraction tree indexing works for every finitary unification theory. The unification algorithm does not need to be implemented in the retrieval algorithm itself and therefore is exchangeable.

3 Path Indexing

3.1 Foundations

In the following we will take a closer look at path indexing which is based on the following idea: Combinations of several path properties are interpreted as restrictions on terms that have to be fulfilled in order to consider a term as being an instance of / unifiable with / a generalization of a query term.

These combinations are stored in *query trees* with one single path property being represented by a *key*. Therefore keys are the foundation of path indexing. In order to define keys properly, we need the notion of *positions*, which will also be introduced.

3.1.1 Positions

A subterm of a term t is described by the position it occurs in t. These positions are defined as sequences of integers, for example the term $h(a)$ occurs at the positions $1 \circ 1$ and 2 in $f(g(h(a)), h(a))$. The subterm of t at position p is denoted by t/p, referring to our example we have $f(g(h(a)), h(a))/1 \circ 1 = h(a)$. The set of all positions of a term is $O(t)$, which will be defined by the next definition following [Ave88] more precisely.

Definition 3.1 (Positions In A Term)
We define the set of all positions of a term $O(t)$ with the help of a special position ε which denotes the empty position. For all terms t we have:

[2] A filter is perfect if it only returns correct answers.

- $t/\varepsilon = t$.

The set of positions $O(t)$ of a term t is recursively defined by:

- $O(t) = \{\varepsilon\}$ *if t is a constant or a variable.*
- $O(t) = \{\ i \circ p \mid 1 \leq i \leq n,\ p \in O(t_i)\} \cup \{\varepsilon\}$ *if $t = f(t_1, \ldots, t_n)$.*

The function $\circ$ represents the concatenation of strings. Note that for every position p we have $\varepsilon \circ p = p \circ \varepsilon = p$.

For example the term $f(a, g(b), x)$ has the positions $O(f(a, g(b), x)) = \{\varepsilon, 1, 2, 2 \circ 1, 3\}$.

3.1.2 Keys

Before we formally define keys, we give an example: The key $[\varepsilon, f, 1, a]$ represents a set of terms having f as the top symbol and the first parameter being equal to a. Keys never contain variables, as all variables in keys are replaced by the same special symbol $*$.

Definition 3.2 (Keys)
A vector of pairs (p_i, s_i) with p_i being a natural number and s_i being a non–variable symbol is called key. Keys are denoted by $[p_1, s_1, \ldots, p_n, s_n]$.

For reasons of simplicity we often do not write the first position p_1, as it happens to be the empty position ε for every key. Also we renounce commas if the result is unambiguous. Our next definition uses the function *top* which returns the top symbol of a term.

Definition 3.3 (Key Fits Term)
A key $k = [p_1, s_1, \ldots, p_n, s_n]$ fits a term t iff all of the following conditions hold:

- $[p_1, s_1, \ldots, p_{n-1}, s_{n-1}]$ *fits* t *if* $n > 1$.
- $p_1 \circ \ldots \circ p_n \in O(t)$.
- $s_n = *$ *if* $t/p_1 \circ \ldots \circ p_n$ *is a variable.*
- $s_n = top(t/p_1 \circ \ldots \circ p_n)$ *if* $t/p_1 \circ \ldots \circ p_n$ *is not a variable.*

In figure 4 we give an example in order to illustrate the definitions introduced above. We chose the tree representation of terms. As another example the key $[f2b]$ fits the terms $f(a, b)$, $f(h(a), b)$, $f(x, b)$, $f(f(a, c), b)$, and so on.

During creation of the path index a total ordering on keys will be needed in order to sort entries which hash to the the same value. This ordering is introduced now.

Definition 3.4 (Comparison Of Keys)
Let $k_1 = [p_{1,1}, s_{1,1}, \ldots, p_{1,n}, s_{1,n}]$ and $k_2 = [p_{2,1}, s_{2,1}, \ldots, p_{2,m}, s_{2,m}]$ be the two different keys. Two keys k_1 and k_2 are equal, written $k_1 = k_2$, iff they are equal in length and all positions and all symbols of the keys are pairwise equal. In a more formal way this is described by

- $n = m \wedge \forall i(1 \leq i \leq n : p_{1,i} = p_{2,i}) \wedge \forall i(1 \leq i \leq n : s_{1,i} = s_{2,i})$

We say a key k_1 is smaller than k_2, written $k_1 < k_2$, iff one of the following conditions is fulfilled.

- $n < m$
- $n = m \wedge \exists i(1 \leq i \leq n : p_{1,i} < p_{2,i} \wedge \forall j(1 \leq j < i : p_{1,j} = p_{2,j}))$
- $n = m \wedge \exists i(1 \leq i \leq n : s_{1,i} < s_{2,i} \wedge \forall j(1 \leq j < i : s_{1,j} = s_{2,j})) \wedge \forall k(1 \leq k \leq n : p_{1,k} = p_{2,k})$

By the way, symbols are compared by considering their internal representations which refer to natural numbers.

For the keys of our example we have: $[f] < [f1a] < [f2g] < [f3*] < [f2g1b]$.

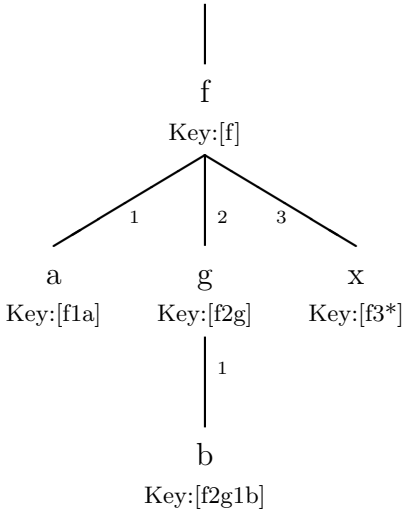

Figure 4: Keys of the term $f(a, g(b), x)$

3.2 Query Trees

3.2.1 Indexing Unifiable Terms

We now consider the properties of a term t unifying a query term s. Obviously, a variable is unifiable with every term, therefore finding unifiers of the variable s is fairly simple. As we describe unifiable terms by restrictions, no restriction is needed in order to represent unifiers of a variable. More difficult is the search for unifiers of a non-variable term s. As easily can be seen, a term s unifies with every variable. Additionally, $s = f(s_1, \ldots, s_n)$ unifies with every term $t = g(t_1, \ldots, t_n)$ which has the same top symbol as s (i.e. $f = g$) and whose arguments t_i unify with the arguments s_i of s. Figure 5 sums up the restrictions on unifiers of a term s. The nodes of the tree are called OR-node and AND-node, respectively.

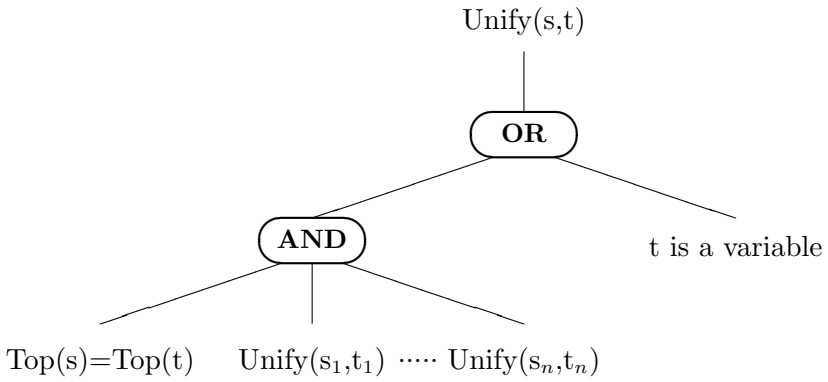

Figure 5: Properties of a term t unifying the non-variable term s

The question that arises is whether really all terms having the properties depicted in figure 5 are unifiable with the query term. Although a set of terms which follows the restrictions contains all unifiable terms, we additionally have to perform occur checks and detect indirect clashes in order to avoid incorrect retrievals. Therefore these restrictions are not a perfect filter. To put it in other words: Terms that fulfill the restrictions are candidates for unifiers only.

Taking a closer look at figure 5 we find out that only two path properties are involved. On the one hand we have to be able to check whether a term is a variable, and on the other hand we are supposed to decide whether a term has a special top symbol. These tasks can easily be solved by keys.

Combining recursive application of the restrictions and the use of keys in order to describe path properties we result in a query tree.

Definition 3.5 (Query Tree)
A tree b is a query tree iff the following conditions hold:

- *All leaves are marked with a key.*
- *All other nodes are marked with either OR or AND.*

Terms that fulfill the restrictions of the query tree in figure 6 are candidates for unifiers of the non-variable term $f(a, g(b), x)$.

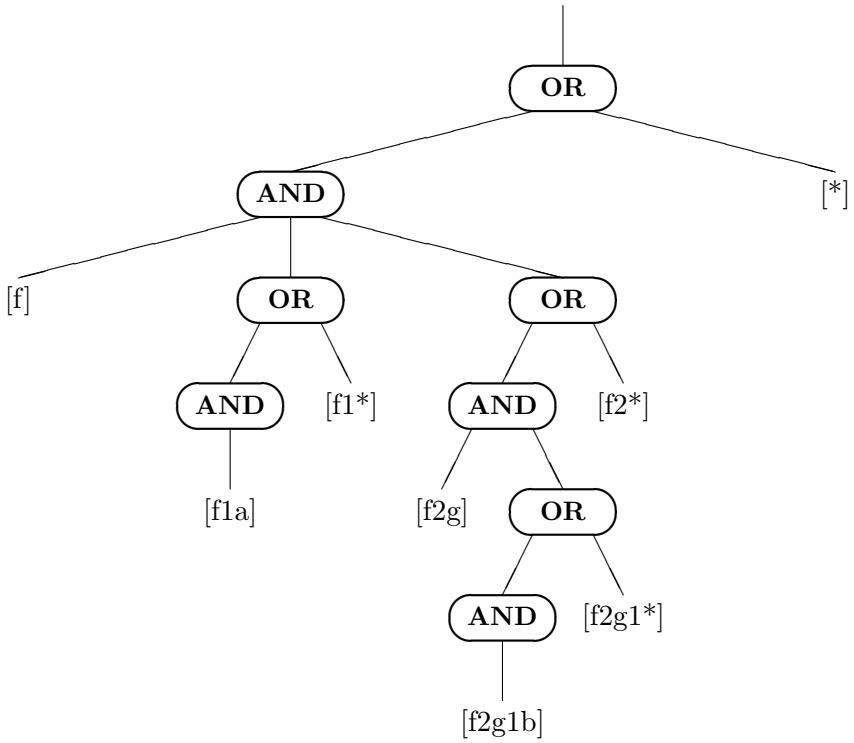

Figure 6: Complete query tree for candidates of unifiers of $f(a, g(b), x)$

We have mentioned that unifiers of variables do not have to be restricted in any way. This is the reason for the lack of a subtree for the variable x in the query tree.

If the indexing depth had been restricted to 2, for example, the keys $[f2g1b]$ and $[f2g1*]$ would not have been taken into consideration. Therefore, reducing the indexing depth causes no additional problems except that more wrong terms pass the filter.

Obviously, the complete query tree depicted in figure 6 contains redundant information and superfluous nodes. We now come up with an algorithm that significantly simplifies the query tree, which leads to faster retrieval of candidates of unifiers of a query term.

Algorithm 3.6 (Query Tree Simplification)
Apply the following four rules once to the complete query tree in the given order.

(1) Skip checking the top symbol of a term $s = f(s_1, \ldots, s_n)$ if f is not a constant and f has at least one argument s_i which is not a variable.

(2) Delete keys which do not fit entries of the database.

(3) Delete an AND-node and all its subtrees if one of the subtrees is empty.

(4) Delete AND-nodes and OR-nodes with less than two successors. Start doing this from the leave nodes of the query tree.

(5) Join OR-nodes not having an AND-operator in between.

Application of the algorithm to the query tree in figure 6 leads to the deletion of the keys [f] and [f2g]. We will now explain why the deletion is correct by using the example of key [f].

$s = f(a, g(b), x)$ has the non-variable arguments a and $g(b)$. We consider the first argument a. In order to describe unifiable terms we use the keys [f1a] and [f1*]. For $g(b)$ we need the keys [f2g1b] and [f2g1*]. As keys include all symbols of the term from the top down to a specific position, the unifiers of a and $g(b)$ will have f as a top symbol anyway.

For the same reason the keys [f1a] and [f2g1b] were not deleted. Obviously, the constants a and b do not have any arguments at all which implies that they do not have any non-variable argument. Assuming that all keys fit some entries of the database, the simplified query tree is depicted in figure 7.

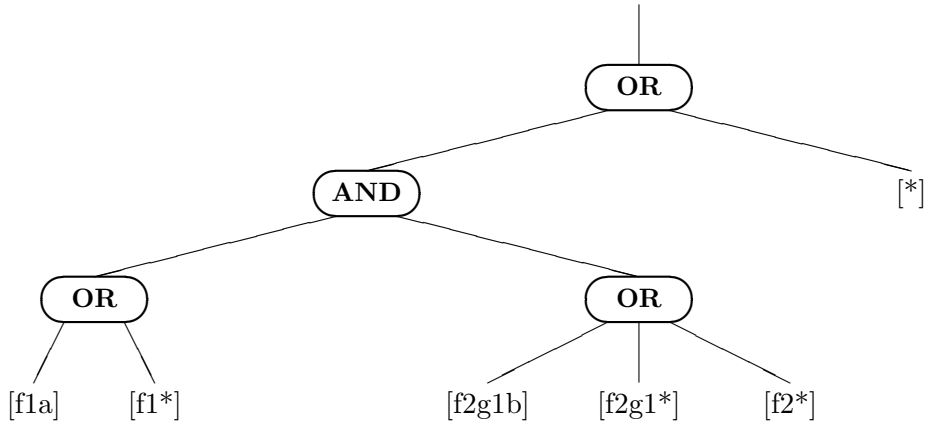

Figure 7: Simplified query tree for candidates of unifiers of $f(a, g(b), x)$

3.2.2 Indexing Instances

Finding instances of a query term t is very simple. We have to distinguish between two cases. On the one hand t could be a variable. In this case we would have to return all entries of our index. On the other hand instances of a non-variable query term t must have the same function symbols as t at the same positions. Variables do not have to be considered. Figure 8 shows the query tree for candidates of instances of $f(a, g(b), x)$.

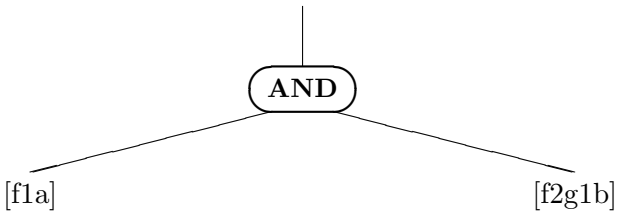

Figure 8: Query tree for candidates of instances of $f(a, g(b), x)$

The query tree could have been derived from the query tree for unification by not regarding keys containing the special symbol $*$ and by applying the simplification algorithm. This results in taking the leaves of the query tree not having a $*$ in their key and combining them with an AND–node. Again the query tree does not represent a perfect filter, occur check failures and clashes can still happen.

3.2.3 Indexing Generalizations

The most complex operation on the index is the search for generalizations of a query term. It is well known that if a term is a generalization of a query term, there exists a matcher from the term to the query term.

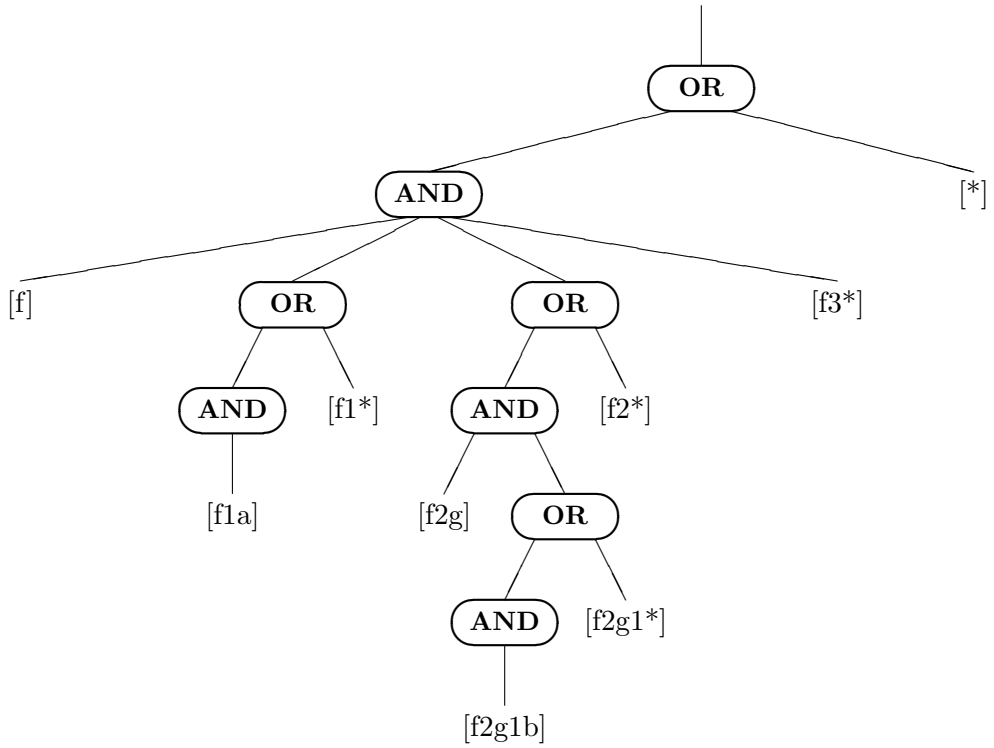

Figure 9: Query tree for candidates of generalizations of $f(a, g(b), x)$

We derive the query tree for finding generalizations of a term by modifying the query tree for unifiable terms of a query term. Remember that we were not able to name any restrictions for unifiers of a variable. This is not the case for generalizations of a variable which will obviously have to be variables, too. Therefore query trees for generalizations of a query term can be derived from query trees for unifiers of a query term by simply adding the key containing the variable restriction to the appropriate AND-node. As an example we show the query tree for candidates of generalizations of $f(a, g(b), x)$ in figure 9. Simplifying the query tree according to algorithm 3.6 is left to the reader.

3.3 Query Tree Evaluation

We have seen that a set of entries of our index can be described by the means of a query tree. The next step towards an efficient retrieval algorithm is the evaluation of the tree information.

Every leaf of a query tree contains a key. The key represents a set of entries sharing a special path property. A query tree is a complex description of a set of entries of an index. These entries can be referred to by their addresses in the system storage.

The main idea of the query tree evaluation function *eval* can be described in just one sentence: To avoid the computation of unions and intersections of lists of addresses, the query tree is constructed and addresses are bubbled up to the root.

Definition 3.7 (Query Tree Evaluation)
Let b be a query tree and let $b_0, \ldots, b_{n-1}$ be the subtrees of b. The query tree evaluation function eval works on a query tree b and an address adr and is recursively defined.

- $eval: \ QUERYTREE \times ADDRESS \longrightarrow ADDRESS$;

ADDRESS is the set of all addresses[3] *of entries in the index unified with the address* `NULL`. `NULL` *is smaller than any other address. The function min returns the smallest address in a set. Note that* $min(\emptyset) =$ `NULL`. *Depending on the operator of the root node of b, we have to distinguish between three cases:*

- *The query tree has a key k in its root:*

  $eval: \ b, adr \mapsto min(\{x | \, x \geq adr \wedge$ *k fits the entry pointed to by* $x\})$;
- *The query tree has the OR-Operator in its root:*

  $eval: \ b, adr \mapsto min(\{eval(b_0, adr), \ldots, eval(b_{n-1}, adr)\} \backslash \{\texttt{NULL}\})$;
- *The query tree has the AND-Operator in its root:*

  $eval: \ b, adr \mapsto min(\{x | \, x \geq adr \ \wedge \forall i (0 \leq i < n : eval(b_i, x) = x)\})$;

Obviously, execution of $eval(b, adr)$ *will lead to the smallest address x which is bigger than adr. Besides, x refers to an entry of the index having the path properties described by the query tree b. Be careful, this is just a definition and not an effective algorithm.*

To achieve fast access to the entries, the ordering of the entries becomes essential. (We have introduced a total ordering on keys, which is not the subject of discussion here). We use the virtual addresses of the entries as the sorting criterion. This allows fast comparison of entries which will be needed during search, insertion, and deletion in the index.

Keys as the leaves of a query tree are the sources of these addresses. It seems reasonable to interpret a key no longer as a set of entries but as a sorted list of addresses to entries of the path index sharing a special path property. As the lists are sorted, the *eval* function performed on a single leaf can easily return the smallest address in its list being greater than or equal to *adr*. If no such address can be found, `NULL` will be returned to the caller.

The AND-nodes in the query tree are the reason for the ordering being of such a substantial importance. In contrast to OR-nodes, which only take the results of their recursive calls and compute the minimum of it, the computations in AND-nodes are less simple. Here we cannot just compute a minimum from a set of addresses, but we have to find the smallest address of a term that has all the properties described by the subtrees of the AND–node. The search for the minimum works as described in the next algorithm:

Algorithm 3.8 (AND-Node Evaluation)
Let b be a query tree and let $b_0, \ldots, b_{n-1}$ *be the n subtrees of b. The query tree has the AND-Operator in its root. 'Minimum' is a variable containing the smallest address which may be accepted.*

(1) $Equal := 0; \ i := 0;$

(2) *while* $(Equal < n)$ *do*

$Answer := eval(b_i, Minimum)$

$if \ (Answer = \texttt{NULL})$ *return* `NULL`

$elseif \ (Answer = Minimum) \ Equal := Equal + 1$

$elseif \ (Answer > Minimum) \ Equal := 1; \ Minimum := Answer$

$i := (i + 1) \ mod \ n$

(3) *return Answer*

Note that 'Minimum' is monotonously increased.

[3] Addresses are natural numbers

3.4 Considering Subterms

The are applications which do not only need to index terms at their top level. If we take paramodulation or the search for critical pairs in a completion procedure into account we notice, that for some applications it might be very useful to consider also the subterms of a term. The way we achieve this is very simple. Take the term $f(a, g(b), x)$, for example, and suppose we want to insert it into an index which also considers subterms. Obviously, we do not only have to insert the term $f(a, g(b), x)$ itself, but also its non-variable[4] subterms a and $g(b)$. Note that the insertion of $g(b)$ leads to the insertion of b also. The greatest advantage of this technique is that the retrieval procedure is not affected. However, insertion time is increased significantly depending on the depth of the terms which are inserted.

3.5 Implementation

Before a query can be started the query tree has to be constructed. During retrieval pointers to entries of the index are bubbled up from the leaves to the root of the query tree. Our implementation of query trees includes some extra features: The leaves of a query tree are represented by a cursor on a list of entries. Every time a query reaches a leaf node of the query tree, the pointer of the entry referred to by the cursor is returned to the caller and the cursor is moved to the next entry. If no more entries can be found `NULL` is returned. We can proceed in that way, because the adr values in calls like $eval(b, adr)$ are monotonously increased for a subtree b. OR-nodes have exactly two subtrees, which eases the search for the minimum pointer to be returned. OR-nodes evaluate their two subtrees and return the smaller value to the caller. The other value which has not been returned is stored and taken into account during the next evaluation of the OR-node. Note that this procedure avoids repeated evaluation of the same subtree with the same adr in calls like $eval(b, adr)$. AND-nodes may have arbitrarily many subtrees. When we create the query tree we apply the query tree simplification algorithm during the construction of the tree.

The path index itself consists of a hash table and a set of secondary lists. With the help of a hash function and a key k we find the corresponding *header* of a list of *connectors*. Collisions are solved by chaining. Headers occurring in the same collision class are kept sorted with the sorting criterion being the total ordering on keys introduced in definition 3.4. All entries found in the list of connectors belonging to a header share the same path property k and are kept sorted using their virtual addresses as the sorting criterion. In figure 10 we see a fragment of the index. As can easily be seen by the example of $f(x, b)$, entries are stored just once in order to save memory.

Let us assume we would like to retrieve entries with the top symbol f and the first argument a. These path properties are represented by the key $[f1a]$. We apply the hash function to that key and get a pointer to a list of headers. Scanning the headers for the right path list leads to the first connector which points to our first solution $f(a, g(x, y))$. In order to avoid the same answer when this query is repeatedly asked, we move the cursor stored in the leaf of our query tree. Continued search leads to $f(a, b)$. Finally, the query tree is deleted.

4 An Interface To Indexing

This section contains a set of functions which represents the interface to indexing in our implementation.

4.1 Index Creation And Deletion

INDEX Index_Create(TYPE);

Before an index can be used it has to be created. `TYPE` has to be one of the two constants `PATH` and `PATH_SUB` where `PATH` stands for ordinary path indexing and `PATH_SUB` will also consider the

[4]Variables are omitted, because one never paramodulates into variables and critical pairs are not created by overlappings into variables, respectively.

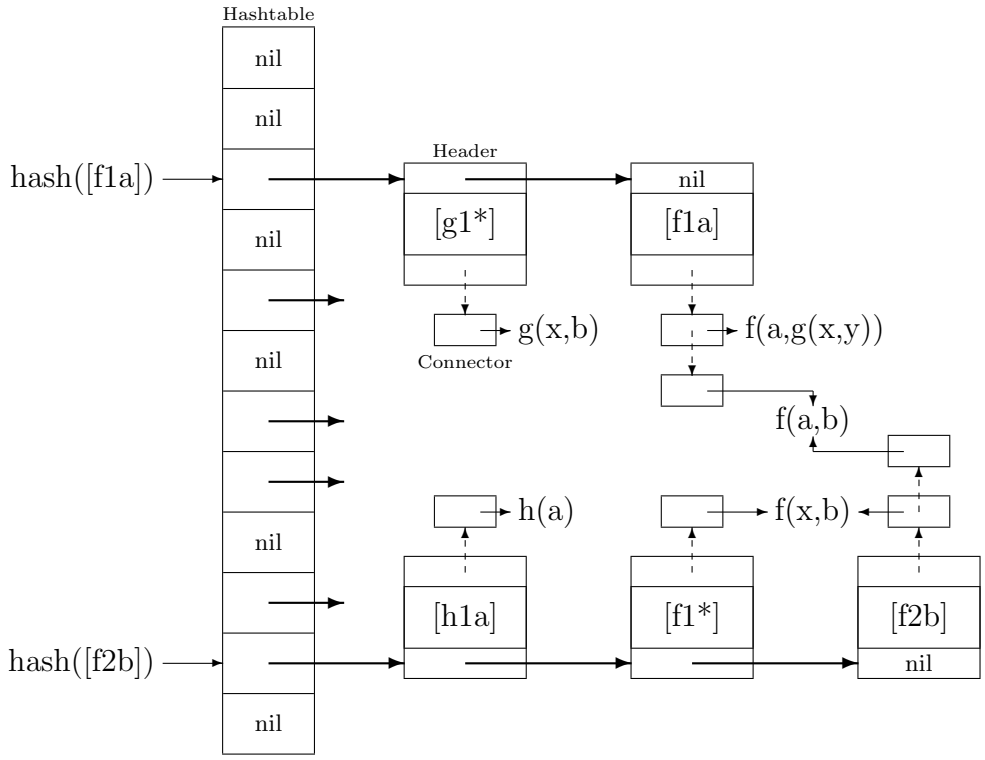

Figure 10: Path Index

subterms of a term. The returned value has to be assigned to a variable of type INDEX which will be needed as a parameter in the following functions.

void Index_Delete(INDEX);

This function deletes an index. The entries of the index itself remain untouched.

void Index_CreateEntry(INDEX,POINTER,TERM);

Adds an entry to the index. The entry is pointed to by the pointer and is supposed to be retrieved under the query term stored in the last parameter.

void Index_DeleteEntry(INDEX,POINTER,TERM);

Removes indexing data structures for the entry pointed to by the pointer using the query term stored in the last parameter.

4.2 Index Retrieval

When using `PATH_SUB`, the retrieval functions do not return a single pointer to your entry, but a pointer to the data structure `INFO`. This structure consists of two entries, of which one is the original pointer you wanted to store and the second is a string which describes the subterm's position in the original term. Use the functions `char* Index_InfoPath(INFO)` and `POINTER Index_InfoPointer(INFO)` to extract that information from `INFO`.

QUERY Index_QueryPrepare(INDEX,MODE,TERM);

With the help of this function we define a query. First we determine the index which is being used during retrieval. Mode contains one of the constants `UNIFIER`, `GENERALIZATION`, or `INSTANCE`. The last parameter contains the query term.

POINTER Index_QueryRetrieve(QUERY);

Returns a pointer to an entry of the index which has the desired feature. Note that you must not insert entries to or delete entries from the index between two calls of `Index_QueryRetrieve()` !

void Index_QueryReset(QUERY);

Has to be called when a query is no more being used and the last retrieval resulted in a Pointer which was not equal to `NULL`.

LIST Index_Retrieve(INDEX,MODE,TERM);

This function combines the three functions `Index_QueryPrepare()`, `Index_QueryRetrieve()`, and `Index_QueryReset()`. It returns a list of pointers to entries of the index. The list may be scanned by the three functions `int List_Empty(LIST)`, `POINTER List_Car(List)`, and `LIST List_Cdr(LIST)`. Use `void List_Delete(LIST)` in order to delete the list.

4.3 Future Interface

In the future we will support abstraction trees. We expect a function ...

BOOL Index_CreateEntryIfNotSubsumed(INDEX,POINTER,TERM);

Insert an entry into the index only if the term is not subsumed (neither forward nor backward) by the entries which have been stored to the index yet.

... to perform very well with abstraction tree indexing.

5 Experiments

In this section we compare our implementation of path indexing to the implementation of path indexing in the OTTER theorem prover [McC92]. The term sets used in the experiments were taken from typical OTTER applications. The sets are paired. There is a set of positive literals and a set of negative literals in each pair. Therefore finding generalizations and instances corresponds to performing the subsumption test for the query set. Unifiable terms are searched in order to find resolution partners and to detect unit conflicts.

A detailed explanation on how the test sets were constructed can be found in [McC92]. CL-pos and CL-neg both contain 1000 terms, both EC-pos and EC-neg represent 500 terms, and the sets BOOL-pos and BOOL-neg consist of 6000 terms each.

The experiments were run on a Sun SPARCstation 1+ computer with 16 MBytes of RAM. The size of the hash table was limited to 500.

The designation "Path-n" indicates path indexing to depth n using our new implementation of path indexing. Since Otter provides indexing to an arbitrary depth, indexing using the OTTER implementation is not limited at all (column "Otter"). The column "OTTER-DT" contains the time needed using discrimination tree indexing.

Indexed Set	Query Set	Path-30	Path-6	Path-3	Otter	Otter-6	Otter-3	Otter-DT
CL-pos	CL-pos	11.4	11.1	13.6	16.1	16.3	39.0	0.8
CL-neg	CL-neg	15.7	12.6	9.2	22.3	17.6	26.4	1.2
EC-pos	EC-pos	4.3	4.1	2.6	5.0	5.0	6.5	0.2
EC-neg	EC-neg	8.6	7.5	2.9	10.1	26.0	32.7	0.4
BOOL-pos	BOOL-pos	18.1	18.2	18.1	28.7	28.7	28.7	0.9
BOOL-neg	BOOL-neg	9.9	9.8	9.5	14.5	14.5	14.2	0.6

Figure 11: Generalization Retrieval

The sum of all retrievals using our implementation is 187.2 seconds, the OTTER implementation needed 353.3 seconds for retrieval. This corresponds to an average of 10.4 seconds verse an average retrieval time of about 19.5 seconds. Anyway, both implementations do not have a chance against discrimination tree indexing.

Indexed Set	Query Set	Path-30	Path-6	Path-3	Otter	Otter-6	Otter-3	Otter-DT
CL-pos	CL-pos	1.6	1.6	4.5	16.1	16.3	39.0	0.8
CL-neg	CL-neg	4.8	3.7	5.1	5.7	4.9	19.9	1.2
EC-pos	EC-pos	0.3	0.3	0.6	1.0	1.1	4.3	1.2
EC-neg	EC-neg	1.2	1.4	1.3	1.4	16.6	30.4	3.1
BOOL-pos	BOOL-pos	5.0	5.1	5.0	7.0	7.1	7.1	3.2
BOOL-neg	BOOL-neg	6.7	6.7	6.6	8.8	8.8	9.2	0.7

Figure 12: Instance Retrieval

Another speed-up appears when retrieving instances (figure 12). Here the average retrieval of our implementation takes 3.4 seconds. OTTER's indexing needs an average of 9.1 seconds.

Indexed Set	Query Set	Path-30	Path-6	Path-3	Otter	Otter-6	Otter-3	Otter-DT
CL-pos	CL-pos	10.8	11.8	18.0	30.2	40.2	127.8	12.5
CL-pos	CL-neg	7.9	7.5	14.0	42.3	42.5	94.2	19.8
CL-neg	CL-pos	10.6	10.2	10.7	49.1	49.0	104.3	26.3
CL-neg	CL-neg	15.0	12.7	10.2	26.3	24.2	41.7	6.0
EC-pos	EC-pos	7.4	7.2	4.2	45.8	45.6	42.7	18.4
EC-pos	EC-neg	14.8	14.7	4.7	61.0	60.5	52.1	16.8
EC-neg	EC-pos	4.2	4.2	3.0	57.1	56.8	55.5	27.0
EC-neg	EC-neg	11.0	7.3	2.7	223.3	220.5	220.1	134.2
BOOL-pos	BOOL-pos	18.7	18.6	18.6	33.1	33.1	33.1	8.1
BOOL-pos	BOOL-neg	16.9	16.9	16.4	26.9	26.9	26.7	2.7
BOOL-neg	BOOL-pos	3.2	3.2	3.3	6.0	6.0	5.9	5.4
BOOL-neg	BOOL-neg	9.8	9.9	9.4	14.6	14.5	14.3	0.9

Figure 13: Unifiable Term Retrieval

Figure 13 leads to the most significant result, however, which is the fact that we only need an average of 10.2 seconds for finding unifiable terms, while OTTER's indexing takes an average of 45.8 seconds.

6 Conclusion

We have shown that our implementation of path indexing significantly decreases search time in the cases of retrieving instances and of finding unifiable terms. The current versions of our implementation and this report are available via anonymous ftp (139.19.20.1). For further information send email to graf@mpi-sb.mpg.de !

References

[Ave88] J. Avenhaus. *Reduktionssysteme.* Universität Kaiserslautern, 1988.

[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*, 8(3), 1992.

[Ohl90] 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.

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