MAX-PLANCK-INSTITUT FÜR INFORMATIK

Extended Path–Indexing

Peter Graf Christoph Meyer

MPI–I–94–253 April 1994

Im Stadtwald
D 66123 Saarbrücken
Germany

Author's Address

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

Publication Notes

This report is a version of a paper submitted to the CADE-12 conference.

Acknowledgements

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) some query term. Among the known indexing methods for term retrieval in deduction systems, Path–Indexing exhibits a good performance in general. However, as Path–Indexing is not a perfect filter, the candidates found by this method have still to be subjected to a unification algorithm in order to detect occur–check failures and indirect clashes. As perfect filters, discrimination trees and abstraction trees thus outperform Path–Indexing in some cases. We present an improved version of Path–Indexing that provides both the query trees and the Path–Index with indirect clash and occur–check information. Thus compared to the standard method we dismiss much more terms as possible candidates.

Keywords

Automated Deduction, Term Retrieval, Path-Indexing.

1 Introduction

The performance of a theorem prover can be increased by speeding up the retrieval and maintenance of data referred to by the deduction system [LO80]. In this context we use a tool which provides fast access to a set of terms, a so-called *index*. We distinguish between the following tasks: Find terms in the index which are unifiable with some *query term*, find terms which are instances of the query term, and find terms which are more general. Because of these abilities, indexing is exploited in order to support different tasks in automated reasoning. In order to find resolution partners for a given literal, for example, a theorem prover has to find unifiable literals. Subsumption of clauses can be detected by the retrieval of generalizations or instances of literals of clauses. Even the retrieval of rewrite rules, demodulators, and paramodulants can be accelerated by indexing if the indexing mechanism also supports retrieval in the subterms of the indexed term set.

The importance of the usage of indexing has been shown by the OTTER theorem prover [McC90]. Due to the consequent usage of Path–Indexing [Sti89, McC92] and Discrimination Tree Indexing [McC92, Chr93, BCR93], this prover became one of the most powerful and fastest deduction systems. Additionally, further techniques such as Abstraction Tree Indexing [Ohl90a, Ohl90b], Coordinate Indexing [Sti89], and Codeword Indexing [HN81, WP84] have been introduced in the literature.

Standard Path–Indexing. The retrieval process is determined by the query term, i.e. the term we search partners for, and the set of entries of the index. In the simplest case these entries are terms also. Path–Indexing is based on two different data structures. Each query results in the construction of a new *query tree* which describes complex restrictions on the entries of the index. Query trees depend on the query term. Entries which fulfill these restrictions are candidates for terms unifiable with the query term. In order to find entries with special path properties we use *Path–Lists*. Path–Lists depend on the terms which were inserted into the index. Strictly spoken, our index consists of a set of Path–Lists and a retrieval mechanism using query trees. However, we sometimes refer to a set of Path–Lists as an index.

In order to find terms which are unifiable with the query term $g(a)$, for example, we create a query tree which contains the following information: Terms unifiable with $g(a)$ are either variables or they have g as a top symbol and the argument of g is either a variable or the constant a. Obviously, query trees may contain three different types of nodes: LEAF-nodes, AND-nodes, and OR-nodes. For each of the properties "term is a variable", "top symbol is g and argument is a variable", and "term is $g(a)$" we maintain a Path–List which is a list of pointers to entries of the index. Eventually, we evaluate the query tree by computing unions (at OR-nodes) and intersections (at AND-nodes) of the Path–Lists referred to in the LEAF-nodes.

Note that Path–Indexing does not identify variables during the search. As we store properties of the form "term is a variable" only, it does not detect, for instance, that the term $f(x, x)$ is not unifiable with $f(y, f(y, z))$, because it treats this unification problem as if it had to unify the terms $f(x_1, x_2)$ and $f(y_1, f(y_2, z)))$. As a consequence we have to apply an ordinary unification algorithm to the entries found and the query term in order to detect occur–check failures and indirect clashes in the case of non–linear terms.

However, Path–Indexing provides some features which other indexing techniques do not have at all or are at least very expensive: A Path–Index may be scanned by several query trees in parallel, which allows parallel or recursive processes to work on the same index. Using query trees we get the results one by one and do not need to evaluate the whole tree. Finally, it is possible to insert entries to and delete entries from a Path–Index even when the retrieval is still in progress.

Extended Path–Indexing. Extending Path–Indexing with term constraints, however, can drastically reduce the number of candidates found in a retrieval and as a consequence accelerate retrieval, because fewer unifications have to be performed. The detection of occur–check failures and indirect clashes in a unification problem does not always require the exact knowledge of the structure and the variables of the terms to be unified. Let us again consider the unification of $f(x, x)$ and $f(y, f(y, z))$. Obviously, the two terms cannot be unified, because the two arguments

of $f(x,x)$ are identical while the two arguments of $f(y,f(y,z))$ cannot be unified because of an occur–check failure (y occurs in $f(y,z)$).

We extend Path–Indexing by creating new lists for terms which have identical subterms at specific positions and for terms which have non–unifiable subterms at these positions. Depending on the properties of the query term, we can reduce the number of entries found by omitting those entries which have non–unifiable subterms at a position where the query term has identical subterms and vice versa.

The advantage of this approach is, that the decision in which set to put a term, depends only on the term itself and needs to be done just once for each entry of our Path–Index, no matter how many retrievals will follow.

2 Preliminaries

We use the standard notions for first order logic. Let V and F be two disjoint sets of symbols. V denotes the set of *variable* symbols. 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 $t_i \in T$. Function symbols with arity 0 are called *constants*. For two terms s and t which are identical we write $s = t$. In our examples we use the symbols x, y, and z as variables and the symbols f, g, and h as function symbols. Constants are denoted by a, b, and c.

A *substitution* σ is a mapping from variables to terms represented by the set of pairs $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ with $\sigma(x_i) = t_i$ for $1 \leq i \leq n$. A *unifier* for two terms s and t is a substitution σ 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 cannot be unified. In contrast to *indirect* clashes and *occur–check failures*, a *direct* clash can be detected without considering partial substitutions. For example, the terms $f(a,y)$ and $f(x,b)$ are unifiable and the unifier is $\{x \mapsto a, y \mapsto b\}$. Furthermore, we get a direct clash when unifying $f(a,x)$ and $f(b,y)$, an indirect clash when unifying $f(x,x)$ and $f(a,b)$, and an occur–check failure when unifying $f(x,x)$ and $f(y,g(y))$.

A *position* in a term is a finite sequence of natural numbers and the subterm of a term t at position p is denoted by t/p. We define the set of all positions $O(t)$ of a term t with the help of a special position ε which denotes the empty position and the function $\circ$ which represents the concatenation of natural numbers.

Definition 2.1 *Let $t = f(t_1,\ldots,t_n)$ be a term. The set of positions of term t is recursively defined by $O(t) = \{\varepsilon\} \cup \{i \circ p \mid p \in O(t_i),\ t \notin F_0,\ t \notin V\}$.*

Definition 2.2 *Let p be a position. The set of terms that contain the position p is defined by $\mathcal{T}_p = \{t \mid t \in T, p \in O(t)\}$.*

For example, $O(h(a,g(b),x)) = \{\varepsilon,\ \varepsilon\circ1,\ \varepsilon\circ2,\ \varepsilon\circ2\circ1,\ \varepsilon\circ3\}$. Constants and variables have the position $\{\varepsilon\}$ only.

A vector of pairs (p_i, s_i) with p_i being a natural number and s_i being a function symbol or the special symbol $*$ is called *path*. Paths are denoted by $[p_1, s_1, \ldots, p_n, s_n]$. Paths are an extension to positions. That means, that we simply add a symbol to each of the natural numbers of a position. For example, we can extend the position $\varepsilon\circ1$ to the path $[\varepsilon, f, 1, a]$. This path represents a set of terms having f as the top symbol and the first argument being equal to a. In Path–Indexing paths do not contain variables. All variables in paths are replaced by the same special symbol $*$. 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 path. We also renounce commas if the result is unambiguous. The following definition of sets of terms with special path properties uses the function *top* which returns the top symbol of a term.

Definition 2.3 *Let $p = p_1 \circ \ldots \circ p_n$ be a position, $[p_1, s_1, \ldots, p_n, s_n]$ a path, and t a term. The set of terms that contain the path p is recursively defined: We have $t \in \mathcal{T}_{[p_1,s_1,\ldots,p_n,s_n]}$ iff all of the*

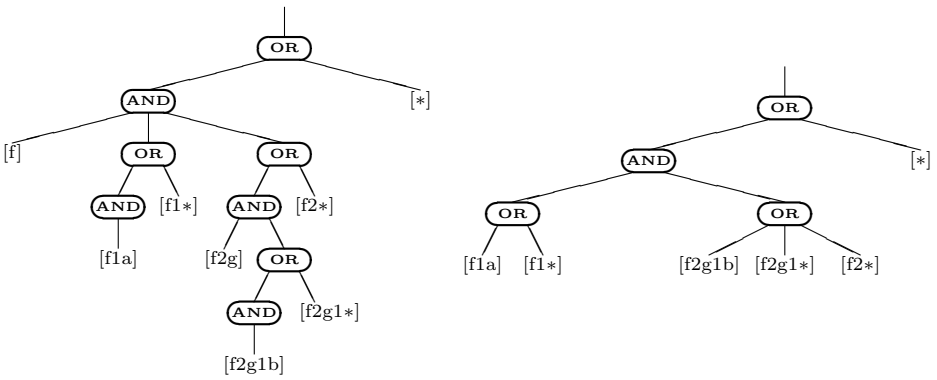


Figure 1: Complete and simplified query tree for the query term $f(a, g(b))$

three conditions hold.
(1) $t \in \mathcal{T}_{[p_1,s_1,\ldots,p_{n-1},s_{n-1}]}$
(2) $t \in \mathcal{T}_{p_1 \circ \ldots \circ p_n}$
(3)

$$s_n = \begin{cases} * & \text{if } t/p \in V \\ \text{top}(t/p) & \text{otherwise} \end{cases} .$$

The terms $h(g(a), b, a)$, $h(x, b, c)$, $h(f(a, c), b, x)$, for example, are all members of the set $\mathcal{T}_{[h2b]}$. The term $h(a, g(b), x)$ is a member of the sets $\mathcal{T}_{[h]}$, $\mathcal{T}_{[h1a]}$, $\mathcal{T}_{[h2g]}$, $\mathcal{T}_{[h2g1b]}$, and $\mathcal{T}_{[h3*]}$.

3 Standard Path–Indexing

Although we will introduce Path–Indexing for the retrieval of terms which are unifiable with some query term, this indexing technique is also able to support the search for instances and more general terms. In this paper, however, we are not going to focus on these features.

Path–Indexing is based on two different data structures. Each query results in the construction of a new *query tree* and in order to find entries with special path properties we use *Path–Lists*.

Query Trees. Query trees describe complex restrictions on the entries of the index. They depend on the query term and are built for non-variable[1] query terms only. A term unifiable with $t = f(t_1, \ldots, t_n)$ either is a variable or it is a complex term $s = f(s_1, \ldots, s_n)$ and the arguments t_i and s_i are pairwise unifiable. We recursively apply this idea to the arguments of t and get a complex tree whose leaves are marked with paths. Consider, for example, the first (complete) query tree in Fig. 1 which represents the restrictions that terms unifiable with the non-variable query term $f(a, g(b))$ have to fulfill.

Obviously, our complete query tree contains superfluous nodes. For example, it is not necessary to check the path [f], because all paths contained in the other subtrees of the corresponding AND-node start with the function symbol f. Additionally, we do not need AND- or OR-nodes if there is only a single subtree below. Therefore, the complete query tree can be simplified. Note that a simplified query tree does not depend on the data in the index. However, we might still be able to reduce the size of the tree if we take the current entries of the index into consideration. Assume that there were no entries with paths $[*]$, $[f2g1b]$, and $[f2g1*]$. The minimal query tree one can get under this assumption is depicted in Fig. 2.

Path–Lists. The index itself consists of a list of secondary indexes, so–called *Path–Lists*. A Path–List is a list of pointers to terms that share a special path property. To find a specific Path–List, we use a hashing mechanism. Figure 2 shows an example of an index. Path–Lists are sorted, which is necessary in order to compute AND-nodes in the query tree efficiently. In our implementation we used the addresses of the terms as the sorting criterion.

[1] A variable x is unifiable with every term t if $x \notin V(t)$ and therefore finding unifiable entries does not result in a restriction on the terms of the index, because variables are not identified in Path–Indexing and therefore all entries of the index have to be found.

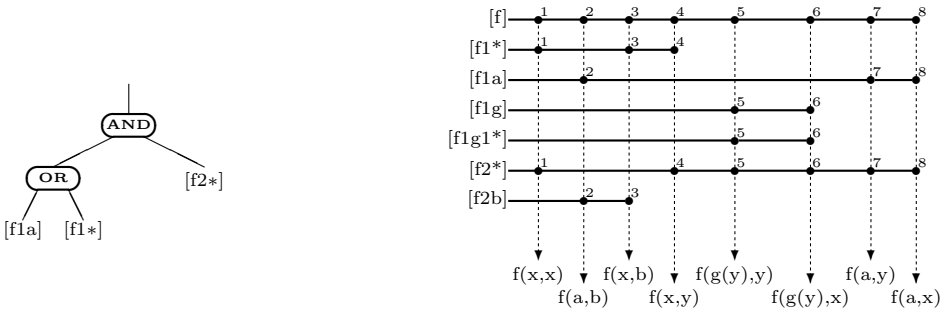


Figure 2: Minimal query tree for $f(a, g(b))$ and Path–Index

Retrieval. Retrieval brings together query trees and Path–Lists. Each leaf of the query tree maintains a cursor on the appropriate Path–List which is initialized by a pointer to the first element of the list. The query tree is evaluated by a recursive algorithm. Application of the algorithm to the root node of the query tree yields either an entry of the index or `NULL`, i.e. there are no more entries left.

Each time a leaf is evaluated, the pointer to the current entry is returned and the cursor is moved one position further. An OR-node returns the minimum of the results of the recursive calls on its subtrees. The computations in AND-nodes are most complex. We have to find the smallest address of a term that has all the properties described by the subtrees of the AND-node [Gra92], i.e. the address has to be found in all subtrees of the AND-node.

We evaluate the query tree of Fig. 2: We start the evaluation of the query tree at the AND-node, which yields two calls. One for the OR-node and one for [f2∗]. The OR-node causes the evaluation of [f1a] and [f1∗]. The LEAF-nodes are evaluated, their cursors are moved, and the terms 2 and 1 are returned. The OR-node computes the minimum and returns 1 to the AND-node. The evaluation of the LEAF-node [f2∗] also results in 1 and therefore $f(x, x)$ is the result of all subtrees of the AND-node and the first result of our retrieval. The next evaluation of the query tree is less simple, because the OR-node returns 3 as a result (the minimum of 3 and 7) while the evaluation of [f2∗] results in 4. Now the AND-node again calls the OR-node, because it returned the smaller value which finally answers 4 also. Therefore $f(x, y)$ is our next result. In the same way we find $f(a, y)$ and $f(a, x)$.

4 Non–Unifiability

The notion of non–unifiability is most important when extending Path–Indexing in order to make it a better filter. In this context we have to answer the question of how we can derive two sets of information from two different terms such that the union of the information sets results in a test for non–unifiability of the terms. As we talk about indexing terms it is of fundamental importance that the information set for each term which is inserted into the index can be computed at insertion time. During retrieval, we compute the information set for the query term once and compare it with the information sets of the terms in the index. In order to define non–unifiability, we need some new notions.

Definition 4.1 *Let p and q be positions. The set of terms that contain the two different positions p and q is defined by $\mathcal{T}_{p,q} := \{\, t \mid t \in \mathcal{T}_p,\, t \in \mathcal{T}_q,\, p \neq q \}$.*

We have three disjoint subsets of $\mathcal{T}_{p,q}$.

Definition 4.2 *The set of terms that contain identical subterms at two different positions p and q is defined by $\mathcal{T}^{=}_{p,q} := \{\, t \mid t \in \mathcal{T}_{p,q},\, t/p = t/q \}$. The set of terms that contain non–unifiable subterms at two different positions is defined by $\mathcal{T}^{\neq}_{p,q} := \{\, t \mid t \in \mathcal{T}_{p,q},\, \neg\exists\sigma \,.\, \sigma(t/p) = \sigma(t/q) \}$. The set of remaining terms is $\mathcal{T}^{*}_{p,q} := \mathcal{T}_{p,q} \setminus (\mathcal{T}^{=}_{p,q} \cup \mathcal{T}^{\neq}_{p,q})$.*

Lemma 4.1 *$\mathcal{T}^{=}_{p,q}$, $\mathcal{T}^{\neq}_{p,q}$, and $\mathcal{T}^{*}_{p,q}$ are disjoint and $\mathcal{T}^{=}_{p,q} \cup \mathcal{T}^{\neq}_{p,q} \cup \mathcal{T}^{*}_{p,q} = \mathcal{T}_{p,q}$.*

Table 1: Some examples of NU

s	t	NU(s,t)	s	t	NU(s,t)
$f(a,b)$	$g(y)$	TRUE	$f(x,g(x))$	$f(g(y),y)$	TRUE
$f(x,x)$	$f(g(y),y)$	TRUE	$f(x,g(x))$	$f(y,g(y))$	FALSE
$f(x,x)$	$f(g(y),g(y))$	FALSE	$f(x,f(x,y))$	$f(f(u,v),f(v,u))$	TRUE
$f(x,x)$	$f(f(y,y),f(a,b))$	TRUE	$h(a,x,x)$	$h(y,y,b)$	FALSE[2]
$f(x,x)$	$f(a,b)$	TRUE	$f(x,b)$	$f(g(x),b)$	FALSE[3]

Additionally, we define three disjoint subsets of $\mathcal{T}^{\neq}_{p,q}$.

Definition 4.3 *The set of terms with non–unifiable variable–term pairs at positions p and q is defined by $\mathcal{T}^{\neq v}_{p,q} := \{\, t \mid t \in \mathcal{T}^{\neq}_{p,q},\, t/p \in V\}$. The set of terms with non–unifiable, non–variable subterms at positions p and q is defined by $\mathcal{T}^{\neq t}_{p,q} := \{\, t \mid t \in \mathcal{T}^{\neq}_{p,q},\, t/p \notin V,\, t/q \notin V\}$.*

Lemma 4.2 *$\mathcal{T}^{\neq v}_{p,q}$, $\mathcal{T}^{\neq v}_{q,p}$, and $\mathcal{T}^{\neq t}_{p,q}$ are disjoint and $\mathcal{T}^{\neq v}_{p,q} \cup \mathcal{T}^{\neq v}_{q,p} \cup \mathcal{T}^{\neq t}_{p,q} = \mathcal{T}^{\neq}_{p,q}$.*

The set $\mathcal{T}^{=}_{p,q}$ consists of two disjoint subsets.

Definition 4.4 *The set of terms with equal variables at positions p and q is defined by $\mathcal{T}^{=v}_{p,q} := \{\, t \mid t \in \mathcal{T}^{=}_{p,q},\, t/p \in V\}$. The set of terms with equal non–variable subterms at positions p and q is defined by $\mathcal{T}^{=t}_{p,q} := \{\, t \mid t \in \mathcal{T}^{=}_{p,q},\, t/p \notin V\}$.*

Lemma 4.3 *$\mathcal{T}^{=v}_{p,q}$ and $\mathcal{T}^{=t}_{p,q}$ are disjoint and $\mathcal{T}^{=v}_{p,q} \cup \mathcal{T}^{=t}_{p,q} = \mathcal{T}^{=}_{p,q}$.*

Only for the sets $\mathcal{T}^{\neq v}_{p,q}$ and $\mathcal{T}^{\neq v}_{q,p}$ the order of the subscripts p and q is important. For all other notations the order of the subscripts does not matter.

Two terms are non–unifiable if they have two different top symbols or if there are two positions which occur in both terms such that for one term the subterms at these positions are identical and the subterms of the other term at these positions are non–unifiable. In order not only to detect indirect clashes we also consider situations in which the two subterms consist of a variable and a term which contains this variable.

Consider the term $t = f(f(g(a),\, g(a)),\, f(g(x),\, x))$, for example. We have $t \in \mathcal{T}^{=t}_{1\circ1,1\circ2}$, because $g(a)$ occurs at the positions $1\circ1$ and $1\circ2$ in t. Additionally, $t \in \mathcal{T}^{\neq v}_{2\circ2,2\circ1}$, because x occurs in $g(x)$, and finally, $t \in \mathcal{T}^{\neq t}_{1,2}$, because $f(g(a), g(a))$ and $f(g(x), x)$ cannot be unified.

Definition 4.5 *Let s and t be two non-variable terms. The predicate of non–unifiability is defined by*

$$\mathit{NU}(s,t) \;\; \textit{iff} \;\; \mathrm{top}(s) \neq \mathrm{top}(t) \vee \exists k,l \in O(s) \cap O(t)\; ((s \in \mathcal{T}^{=}_{k,l} \;\wedge\; t \in \mathcal{T}^{\neq}_{k,l}) \vee (s \in \mathcal{T}^{\neq}_{k,l} \;\wedge\; t \in \mathcal{T}^{=}_{k,l}) \vee (s \in \mathcal{T}^{\neq v}_{k,l} \;\wedge\; t \in \mathcal{T}^{\neq v}_{l,k})).$$

Note that the predicate serves as a filter only; NU is sufficient for non–unifiability, but not necessary, i.e. NU does not detect all failures. However, NU(s,t) will never be TRUE if the terms s and t are unifiable. Additionally, NU is defined for non-variable terms only. In Table 1 we see some examples.

Obviously, the concept of non–unifiability cannot generally be used in Path–Indexing, because there are far too many combinations of positions to provide a Path–List for each one. We will now restrict the definition of NU in order to get a concept applicable to Path–Indexing.

[2] NU fails, because the indirect clash can only be detected using the information $x = y$, which can be derived when unifying the terms only.

[3] Here the result of NU is not correct, because the sets of variables of s and t are not disjoint. However, this does not cause real problems, because clauses are renamed each time an inference is applied.

Table 2: Connections of the term sets

Query Term	Where to Search	Where NOT to Search
$t \in \mathcal{T}^{=}_{p,q}$	$s \in \mathcal{T}^{*}_{p,q} \cup \mathcal{T}^{=_v}_{p,q} \cup \mathcal{T}^{=_t}_{p,q}$	$s \notin \mathcal{T}^{\neq_v}_{p,q} \cup \mathcal{T}^{\neq_v}_{q,p} \cup \mathcal{T}^{\neq_t}_{p,q}$
$t \in \mathcal{T}^{\neq_v}_{p,q}$	$s \in \mathcal{T}^{*}_{p,q} \cup \mathcal{T}^{\neq_v}_{p,q} \cup \mathcal{T}^{\neq_t}_{p,q}$	$s \notin \mathcal{T}^{=_v}_{p,q} \cup \mathcal{T}^{=_t}_{p,q} \cup \mathcal{T}^{\neq_v}_{q,p}$
$t \in \mathcal{T}^{\neq_t}_{p,q}$	$s \in \mathcal{T}^{*}_{p,q} \cup \mathcal{T}^{\neq}_{p,q}$	$s \notin \mathcal{T}^{=_v}_{p,q} \cup \mathcal{T}^{=_t}_{p,q}$
$t \in \mathcal{T}^{*}_{p,q}$	$s \in \mathcal{T}_{p,q}$	$s \notin \emptyset$

5 Extended Path–Indexing

Standard Path–Indexing uses query trees as complex descriptions of restrictions on terms. We will now extend both the query trees and the index itself by not only checking direct clashes as in ordinary Path–Indexing but also considering non–unifiability. More precisely, we add non–unifiability restrictions to the query trees which consider indirect clashes and occur–check failures. The index, however, remains a filter only, although there will be more lists in the index. We still will have to apply an ordinary unification algorithm in order to test unifiability and to extract the unifying substitutions.

5.1 Adapting Non–Unifiability

We have learned from Standard Path–Indexing that the unification partners for terms with special properties can only be found in special sets. Table 2 sums up the connections between the sets when non–unifiability is taken into account. The column "Where to Search" can directly be derived from the definition of NU. From the Lemmata 4.1, 4.2 and 4.3 we get the third column of the table.

The reason for considering the sets in which a partner for the query term t *cannot* occur is simple: Query trees represent restrictions on the term sets. On the one hand, we will extend the query trees with additional restrictions resulting from non–unifiability which, on the other hand, will be tested with the help of the disjoint sets $\mathcal{T}^{=_v}_{p,q}$, $\mathcal{T}^{=_t}_{p,q}$, $\mathcal{T}^{\neq_v}_{p,q}$, $\mathcal{T}^{\neq_v}_{q,p}$, and $\mathcal{T}^{\neq_t}_{p,q}$. We avoid the usage of $\mathcal{T}^{\neq}_{p,q}$, because $\mathcal{T}^{\neq}_{p,q}$ contains the subsets $\mathcal{T}^{\neq_v}_{p,q}$ and $\mathcal{T}^{\neq_v}_{q,p}$ which have to be created for the case $t \in \mathcal{T}^{\neq_v}_{p,q}$ anyway. Using $\mathcal{T}^{=_v}_{p,q}$ and $\mathcal{T}^{=_t}_{p,q}$ instead of $\mathcal{T}^{=}_{p,q}$ results in better constraints in the query tree.

We have also stated that it is impossible to provide a Path–List for each possible combination of positions in a term. In order to restrict the number of combinations we introduce the notion of *distance* of positions.

Definition 5.1 *Let $p = p_1 \circ \ldots \circ p_n$ and $q = q_1 \circ \ldots \circ q_m$ be two different positions. Let $k = \max\{i \,|\, \forall j.\, 1 \leq j \leq i \leq \min\{n,m\} : p_j = q_j\}$ be the length of the longest common prefix of p and q. The distance of p and q is defined by*

$$\mathrm{dist}(p,q) := \begin{cases} \infty & \text{if } k = \min\{n,m\} \\ \max\{n,m\} - k & \text{otherwise} \end{cases}.$$

For example, $\mathrm{dist}(1\circ2\circ1, 1\circ1\circ2\circ2) = 3$ and $\mathrm{dist}(1\circ2\circ1, 1\circ2\circ2) = 1$. Additionally, we have $\mathrm{dist}(1\circ2, 1) = \infty$, because 1 is a prefix of $1\circ2$. Identical positions also have distance ∞.

Definition 5.2 *Let p and q be two different positions. The set of terms that contain two positions with a maximum distance of $n > 0$ is defined by*
$\mathcal{T}_{p,q,n} := \{\, t \,|\, t \in \mathcal{T}_{p,q}, \mathrm{dist}(p,q) \leq n\}$.
The maximum distance of p and q is also called NU–depth.

In case the NU–depth $n > 0$ is constant, we may drop the subscript and write $\mathcal{T}_{p,q}$ instead of $\mathcal{T}_{p,q,n}$. In the following we regard the definitions of Section 4 as if they had been made using $\mathcal{T}_{p,q,n}$ for a constant NU–depth. Note, that if we had restricted the NU–depth to 1, for instance, the non–unifiability of $f(x, f(x,y))$ and $f(f(u,v), f(v,u))$ would not have been detected.

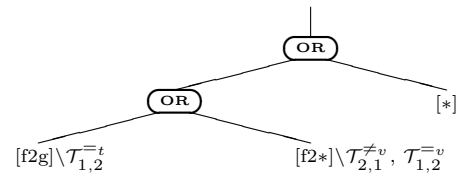
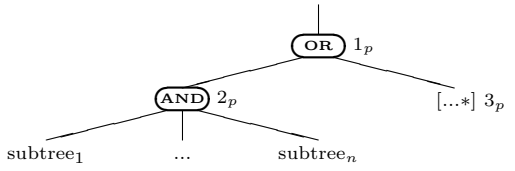


Figure 3: Extended query tree for the query term $f(x, g(x))$, NU–depth=1



	Add Set	Node
$t \in \mathcal{T}_{p,q}^{=}$	$\mathcal{T}_{p,q}^{\neq t}$	2_p
	$\mathcal{T}_{p,q}^{\neq v}$	3_p
	$\mathcal{T}_{q,p}^{\neq v}$	3_q
$t \in \mathcal{T}_{p,q}^{\neq v}$	$\mathcal{T}_{p,q}^{=t}$	2_p
	$\mathcal{T}_{p,q}^{=v}$	3_p
	$\mathcal{T}_{q,p}^{\neq v}$	3_q
$t \in \mathcal{T}_{p,q}^{\neq t}$	$\mathcal{T}_{p,q}^{=t}$	2_p
	$\mathcal{T}_{p,q}^{=v}$	3_p

Figure 4: Add sets to constraints

5.2 Extending the Data Structures

Extended Query Trees. We apply our new definition of non–unifiability to query trees. Actually, there is no big difference between standard and extended query trees, except for some constraints which are added to specific nodes. As an example we present the extended query tree for terms s_i which are unifiable with the term $t = f(x, g(x))$ in Fig. 3. The NU–depth is set to 1. The three constraints $s_i \notin \mathcal{T}_{1,2}^{=v}$, $s_i \notin \mathcal{T}_{1,2}^{=t}$ and $s_i \notin \mathcal{T}_{2,1}^{\neq v}$ are due to the fact that $t \in \mathcal{T}_{1,2}^{\neq v}$ (see Table 2). Constrained nodes affect the evaluation of the query tree in a very simple and efficient way: When addresses of terms are bubbled up from the leaves to the root of the tree, only those addresses pass a constrained node which do not occur in the sets described by the constraint. Note that a constraint may consist of more than one set.

We now focus on the positions of the constraints in the query tree. The retrieved set of terms would not change even if we attached all constraints to the root node of the query tree. However, each constraint has its distinct position in the tree in order to reduce evaluation costs. In Fig. 3 the constraint $s_i \notin \mathcal{T}_{2,1}^{\neq v}$ is attached to the leaf [f2∗], because $\mathcal{T}_{2,1}^{\neq v}$ contains terms with variables at position 2 only. In parallel $\mathcal{T}_{1,2}^{=t}$ which consists of non–variable terms at position 2 only is not searched for variables resulting from leaf [f2∗], when the constraint is attached to the leaf [f2g]. As extended query trees can also be simplified and minimized, we can omit some constraints in case the appropriate node in the query tree is deleted.

Let us take a closer look at where in the query tree the constraints have to be added. Consider Fig. 4, a fragment of the query tree which corresponds to the query term $t \in \mathcal{T}_{p,q}$ at position p. Depending on the properties of t, we have to add sets to constraints at specific positions of the extended query tree. The table in Fig. 4 shows the insertion rules.

Note that the set $\mathcal{T}_{q,p}^{\neq v}$ is added to the constraint of node 3_q of the query tree fragment which represents the query term at position q. Additionally, we will result in less recursive calls when evaluating the extended query tree if we do not add the set $\mathcal{T}_{p,q}^{\neq t}$ to node 2 but to node 1 of subtree$_1$. In Fig. 5 the extended query tree for terms which are unifiable with the query term $f(g(x), g(h(x)))$ is depicted. The NU–depth is set to 2.

Just like in Standard Path–Indexing the query tree is simplified and minimized before the retrieval is started. Again we may delete those LEAF-nodes which correspond to empty Path–Lists. Additionally, we may delete all empty sets in constraints. In case an AND-node or an OR-node is deleted, because it has just one subtree, the constraint is inherited to the root of the subtree. In this way we get a minimal extended query tree which is based on the extended

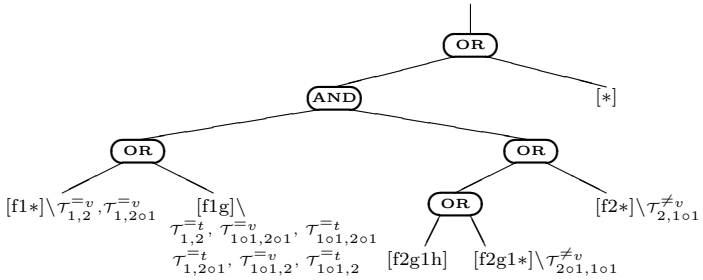


Figure 5: Extended query tree for the query term $f(g(x), g(h(x)))$, NU–depth=2

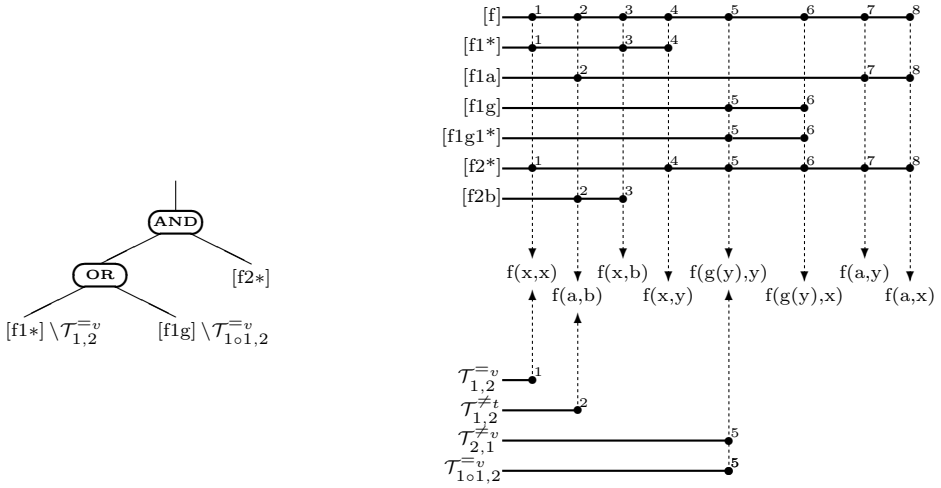


Figure 6: Minimal extended query tree for $f(g(z), g(h(z)))$ and extended Path–Index

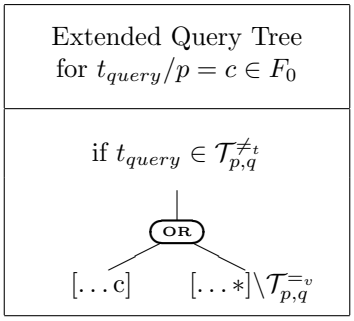
Path–Index of Fig. 6. See Section 6 for further details.

Extended Path–Lists. Path–Lists are created when terms are inserted into the index. In order to provide the Path–Lists for the tests of the constraints, one has to detect for every possible combination of positions p and q of a term s with $\text{dist}(p, q) \leq$ NU-search depth, whether s is in one of the sets $\mathcal{T}^{=v}_{p,q}$, $\mathcal{T}^{=t}_{p,q}$, $\mathcal{T}^{\neq v}_{p,q}$, $\mathcal{T}^{\neq v}_{q,p}$, or $\mathcal{T}^{\neq t}_{p,q}$. In parallel to ordinary Path–Lists the extended Path–Lists are represented as sorted lists. In this context it is important that these lists are sorted according to the same sorting criterion as the ordinary Path–Lists. Note that the insertion of a term does not need to result in an entry in an extended Path–List, see $f(x, b)$ or $f(x, y)$ for example.

Extended Retrieval. The retrieval algorithm which allows for extended query trees is just a slight modification of the ordinary retrieval algorithm. The only difference is the fact that the entry which has been found in a node of an extended query tree is tested against the constraints, i.e. we search for the entry in the appropriate extended Path–Lists. Obviously, the search takes advantage of the ordering of the lists and may start at the position in the list where the last comparison stopped. In case we find the entry in one of the lists, we start a new retrieval on the current node of the tree, which will yield another entry. Otherwise the entry is moved towards the root of the extended query tree.

In the example depicted in Fig. 6 we would find the terms $f(x, x)$, $f(x, y)$, $f(g(y), y)$, and $f(g(y), x)$ if we used Standard Path–Indexing. However, Extended Path–Indexing exploits the constraints and the found terms are $f(x, y)$ and $f(g(y), x)$ only, because $f(x, x)$ is rejected at the OR-node by constraint $\mathcal{T}^{=v}_{1,2}$ and $f(g(y), y)$ is rejected at the LEAF-node [f1g] by constraint $\mathcal{T}^{=v}_{1\circ1,2}$.

Table 3: Constraints for constant subterms



6 Algorithms and Data Structures

6.1 Simplification and Minimalization of Extended Query Trees

In order to explain the insertion rules of constraints in consideration of simplification and minimalization of query trees we focus on the different types of subterms which can occur in a query term. In contrast to the complete insertion rules of Fig. 4 many constraints in an extended query tree can be omitted:

1. Constraints of empty leaf nodes can be left out. For this reason we just have to give insertion rules for non-empty leaf nodes, respectively for non-empty Path–Lists.
2. We do not have to consider all of the five different types of constraints for every kind of subterm in the *query term*, e.g. terms with a constant subterm at position p cannot be element of $\mathcal{T}^{=v}_{p,q}$ because this constraint contains terms with variables at position p only.
3. There are constraints which do not reject *Path–Index terms* because they do not contain any terms which will be retrieved by the associated query tree nodes.

Note that we explicitly refer to *query term* in 2. because a property of the query term prevents constraint insertion in difference to 3. where a property of *Path–Index terms* is responsible for omitting constraints. We introduce three lemmata which can be applied on 2. and 3.

Lemma 6.1 (Constant subterms)
Let t be a term, p a position, and $t/p \in F_0$. Then $\forall q \in O(t) : t \notin \mathcal{T}^{=v}_{p,q} \cup \mathcal{T}^{\neq v}_{p,q} \cup \mathcal{T}^{\neq v}_{q,p}$.

Lemma 6.2 (Variable subterms)
Let t be a term, p a position, and $t/p \in V$. Then $\forall q \in O(t) : t \notin \mathcal{T}^{=t}_{p,q} \cup \mathcal{T}^{\neq t}_{p,q} \cup \mathcal{T}^{\neq v}_{q,p}$.

Lemma 6.3 (Function subterms)
Let t be a term, p a position, and $t/p \in F \backslash F_0$. Then $\forall q \in O(t) : t \notin \mathcal{T}^{=v}_{p,q} \cup \mathcal{T}^{\neq v}_{p,q}$.

In the following t_{query} is a term for which an extended query tree is created. Furthermore, p denotes a position in the query term t_{query}.

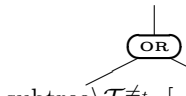
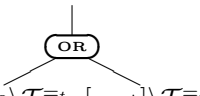
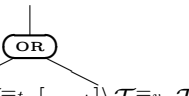
Constraints for constant subterms. Table 3 shows the complete extended query tree for constant subterms. The constraint $\mathcal{T}^{=v}_{p,q}$ has to be added to node $[\ldots *]$ only if $t_{query} \in \mathcal{T}^{\neq t}_{p,q}$. $\mathcal{T}^{=t}_{p,q}$ can be omitted because of the following Lemma:

Lemma 6.4 (Query Trees detect direct clashes of constant subterms)
For query terms with constant subterms at position p the constraints $\mathcal{T}^{\neq v}_{p,q}$, $\mathcal{T}^{\neq v}_{q,p}$, $\mathcal{T}^{=t}_{p,q}$, and $\mathcal{T}^{\neq t}_{p,q}$ do not provide restrictions on retrieved terms.

Obviously, $\mathcal{T}^{\neq_v}_{p,q}$ and $\mathcal{T}^{\neq_v}_{q,p}$ are empty for constant subterms because variables cannot occur in constants. The constraints $\mathcal{T}^{=_t}_{p,q}$ and $\mathcal{T}^{\neq_t}_{p,q}$ do not provide more information on constant subterms as query trees do. A query tree only returns a term with the same constant symbol as in the query term, in other words it detects direct clashes. Therefore a query tree returns terms which are not element of $\mathcal{T}^{=_t}_{p,q}$ or $\mathcal{T}^{\neq_t}_{p,q}$.

Now we focus on the remaining cases for t_{query}. If $t_{query} \in \mathcal{T}^{=_t}_{p,q}$ all constraints $\mathcal{T}^{\neq_t}_{p,q}$, $\mathcal{T}^{\neq_v}_{p,q}$, and $\mathcal{T}^{\neq_v}_{q,p}$ can be omitted because the condition for equal constants is completely solved by the query tree as stated in Lemma 6.4. The remaining cases do not have to be considered because t_{query} cannot be element of $\mathcal{T}^{=_v}_{p,q}$, $\mathcal{T}^{\neq_v}_{p,q}$, or $\mathcal{T}^{\neq_v}_{q,p}$ by Lemma 6.1 with $t_{query}/p \in F_0$. The resulting extended query tree is equal to the standard query tree for constant subterms.

Table 4: Constraints for function subterms

Extended Query Trees for $t_{query}/p = \mathrm{f} \in F \backslash F_0$		
if $t_{query} \in \mathcal{T}^{=_t}_{p,q}$	if $t_{query} \in \mathcal{T}^{\neq_t}_{p,q}$	if $t_{query} \in \mathcal{T}^{\neq_v}_{q,p}$
 subtree$\backslash\mathcal{T}^{\neq_t}_{p,q}$ $[\ldots *]\backslash\mathcal{T}^{\neq_v}_{p,q}$	 subtree$\backslash\mathcal{T}^{=_t}_{p,q}$ $[\ldots *]\backslash\mathcal{T}^{=_v}_{p,q}$	 subtree$\backslash\mathcal{T}^{=_t}_{p,q}$ $[\ldots *]\backslash\mathcal{T}^{=_v}_{p,q}, \mathcal{T}^{\neq_v}_{p,q}$
add $\mathcal{T}^{\neq_v}_{q,p}$ to node at position q		

Constraints for function subterms The extended query trees for function subterms are shown in Table 4. All constraints concerning the insertion rules of Fig. 4 have to be added. Note that the constraint $\mathcal{T}^{\neq_v}_{q,p}$ cannot be added to the query tree directly if t_{query} is element of $\mathcal{T}^{=_t}_{p,q}$. The constraint is saved in a stack until the query tree fragment which represents the query term at position q is created. The query tree fragment for position q is created later because of the way the combinations of p and q are generated. See Algorithm 6.1 for details on generating p, q combinations.

The extended query tree for the arguments of the function subterm is called *subtree*. If the root of *subtree* is an AND–node all constraints which have to be added to *subtree* are inherited to one of the AND–node's sons.

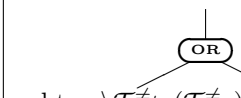
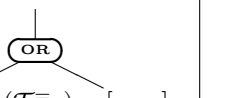
Note that we do not consider $t_{query} \in \mathcal{T}^{=_v}_{p,q}$ or $t_{query} \in \mathcal{T}^{\neq_v}_{p,q}$ because t_{query} cannot be element of these constraints by Lemma 6.3 with $t_{query}/p \in F \backslash F_0$.

Constraints for variable subterms Variables of a query term are not considered for the standard query tree creation, i.e. standard query trees do not contain any nodes with information on query term variables. Even though constraints on variables in the query term can reduce the amount of found entries. These constraints are added to the root of *subtree*. The *subtree* represents the non-variable arguments of the function subterm in which the current variable occurs. If the root of *subtree* is an AND–node the constraints are inherited as mentioned above.

The constraints within brackets are added only if the Path–Index contains terms with variables at position p. Otherwise the retrieved terms are not element of this kind of constraints by Lemma 6.3 or Lemma 6.1 applied on these terms. Furthermore, the constraint $\mathcal{T}^{\neq_v}_{q,p}$ is saved in order to insert it later in the query tree fragment for q.

Note that t_{query} cannot be element of $\mathcal{T}^{=_t}_{p,q}$, $\mathcal{T}^{\neq_t}_{p,q}$, or $\mathcal{T}^{\neq_v}_{q,p}$ by Lemma 6.2 with $t_{query}/p \in V$. Therefore we consider the constraints $\mathcal{T}^{=_v}_{p,q}$ and $\mathcal{T}^{\neq_v}_{p,q}$ only.

Table 5: Constraints for variable subterms

Extended Query Trees for $t_{query}/p = \mathrm{x} \in V$	
if $t_{query} \in \mathcal{T}^{=_v}_{p,q}$  subtree$\backslash\mathcal{T}^{\neq_t}_{p,q}, (\mathcal{T}^{\neq_v}_{p,q})$ $[\ldots *]$	if $t_{query} \in \mathcal{T}^{\neq_v}_{p,q}$  subtree$\backslash\mathcal{T}^{=_t}_{p,q}, (\mathcal{T}^{=_v}_{p,q})$ $[\ldots *]$
Add $\mathcal{T}^{\neq_v}_{q,p}$ to query tree fragment at position q	

Extended Query Tree Construction. The construction of an extended query tree is, relative to its node structure, exactly the same as the construction of a standard query tree. For further details see [Gra92]. The data structure for leaf-nodes and OR-nodes in extended query trees is extended with a pointer to a list of constraints because all constraints are added to leaf-nodes and OR-nodes only. The constraints for AND-nodes are inherited to one of the direct sons which cannot be AND-nodes. Note that all subtrees of a query tree fragment and the fragment itself have been minimized and simplified before constraints are added.

In order to create an extended query tree the query term is scanned in depth first order. The found positions are called p. Additionally, all combinations p, q of positions in the query term have to be enumerated to determine the constraints. In order to avoid symmetric combinations all positions q are enumerated to the "right" of p only. We introduce the required order on positions in the following definition.

Definition 6.1 (Comparison of Positions)
Let $p = p_1 \circ \ldots \circ p_n$ and $q = q_1 \circ \ldots \circ q_m$ be two different positions with $n, m > 0$. We define p is left to q, written $p < q$, as follows:

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

Additionally, if $p < q$ say q is to the right of p.

For example, $1 \circ 1 < 1 \circ 2$ or $1 \circ 1 < 2$, but not $1 \circ 2 \circ 1 < 1 \circ 2$.

Another restriction on position combinations is the NU–depth. Algorithm 6.1 computes the positions to the "right" of a given position p considering the NU–depth:

Algorithm 6.1 (Position Combinations)
Let $t \in T$ be a term and $p = p_1 \circ \ldots \circ p_n$ a position with $p \in O(t)$. Position p is fixed.

1. *Step up term t starting at position p until the root of t or the* NU–depth *is reached. For each step do:*
2. *Visit the remaining arguments which are to the "right" of the current subterm. For each argument do:*
3. *Traverse the argument in depth first order considering the* NU–depth *and index depth. Every found position is located to the "right" of position p and has a distance to position p which is smaller or equal than the NU–depth.*

Algorithm 6.1 tries to find more arguments to the right of each argument on the path of position p. Position p is stepped in reverse direction up to the root, respectively until the NU–depth is reached. Every found argument is immediately searched in depth first order under restriction of

the NU–depth, i.e. the algorithm is allowed to descend in every found subterm NU-1 times. Note that the algorithm can be used either on terms which will be inserted in a constraint index or on terms for which extended query trees will be created.

The constraint sets which have to be integrated in an extended query tree are determined as described in the last three paragraphs. Most of the constraints can directly be integrated during the search of the term and the creation of the extended query tree. The constraint $\mathcal{T}_{q,p}^{\neq_v}$ has to be connected to a node which is associated to position q. Our search strategy guaranties that every "right" position q is reached after position p was attended. Therefore constraints which have to be integrated later are pushed onto a stack. Every time a new position p is created this stack has to be searched for appropriate constraints by applying the order of Definition 6.1. All constraints in the stack having positions to the "left" of the current position p will be deleted. This can happen because of minimalization, i.e. several query tree nodes are left out because of the lack of index entries. Therefore several constraints from the stack are not needed anymore.

6.2 Extended Path–Index

Data Structure. The Extended Path–Index is established as a record of several hash tables. For indexing on path properties of terms we use the same hash table as in Standard Path–Indexing. The hashing has been described in [Gra92]. The constraint sets are saved in similar tables except for the hash key which is built up by the two positions of a constraint. One hash table is used for one kind of a constraint set. Every hash table can be allocated in a different size.

Maintenance of the Extended Index. In order to insert or delete terms, we basically use the same procedures as in Standard Path–Indexing. Additionally, every possible combination of positions p and q of a term has to be enumerated by Algorithm 6.1 to obtain all constraints on this term. As an extention to Path–Indexing respectively to Extended Path–Indexing we provide two different methods for term deletion. Accordingly, terms can be deleted physically or by just marking its entries as deleted. The latter method provides deletion of terms from the index even though query trees exist. The marked terms will not be considered anymore. In order to remove the marked terms entirely from the index a garbage collection procedure is used.

6.3 Retrieval in Extended Query Trees

The retrieval algorithm for extended query trees has to be changed for leaf-nodes and OR-nodes only. AND-nodes do not contain any constraints. The found entries are tested against the constraints which are stored in a linear list. Completely searched constraints will be deleted during retrieval.

6.4 Improving Extended Path–Indexing Data Structures

In this section we introduce some improvements on Extended Path–Indexing which both accelerate retrieval and reduce memory requirements.

Replace the hashing mechanism by a search tree. A major shortcoming of Extended Path–Indexing is its memory requirement. An examination of the calls for memory allocation showed that a very large part of the memory is used for storing paths. As the index depth is limited, paths are implemented as arrays of constant length. However, most of the paths do not reach this length and a lot of memory is allocated but never used. Additionally, many paths share a prefix, but this property is not exploited. Finally, using a hashing mechanism to find the appropriate Path–Lists for a specific path has to scan this path several times. We compute the value of the hash function and have to compare the path with the paths stored in the same collision class. Solving the collisions forces us to store the paths in the hash table. One way to avoid these comparisons and to get rid of storing redundant parts of the paths is to use a tree as depicted in Figure 7.

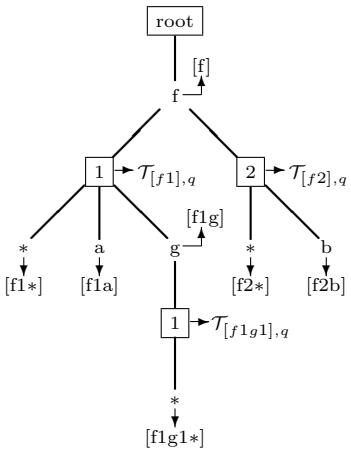


Figure 7: A new way of accessing Path– and Constraint–Lists

Use Paths instead of Positions for Constraints. Although we presented Extented Path–Indexing using positions in order to describe term constraints, one could also exploit the more detailed information of paths in this context. We could speak of $\mathcal{T}^{\neq_v}_{[f1],[f2g1]}$ instead of $\mathcal{T}^{\neq_v}_{1,2\circ1}$, for instance. Using this technique will lead to more but shorter lists for the constraints. On one hand, the retrieval is accelerated, because less entries have to be scanned. On the other hand, it is more time consuming to hash paths instead of positions. Note that the last symbol in the paths is not needed in order to specify a constraint.

Store Constraints together with ordinary paths. Using paths instead of positions to specify constraints can also be exploited in order to ease access to the extended Path–Lists. We integrate the extended Path–Lists into the search tree for paths in the following way: Standard Path–Lists are attached to nodes which contain symbols only. The nodes in Fig. 7 which contain the number of the argument (written in boxes) can be used to lead to further information. Such a box represents all but the last symbol of a path p. At this node we can attach information concerning path p: For each of the constraint sets $\mathcal{T}^{=_v}_{p,q}$, $\mathcal{T}^{=_t}_{p,q}$, $\mathcal{T}^{\neq_v}_{p,q}$, and $\mathcal{T}^{\neq_t}_{p,q}$ we maintain a tree which leads to the corresponding sets for a specific path q. The main advantages are very small memory requirements and the fact that the first path of a constraint has to be scanned just once. For all constraints which have to be taken into consideration for this path we have to focus on the second path only. Additionally, when traversing the tree in order to find Standard Path–Lists we pass these nodes anyway. No additional search has to be done for the first path.

7 Experiments

Term Sets. For our experiments we used the term sets which were introduced in [McC92]. The term sets 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. 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 $g(x,g(g(g(g(B,B),y),z),u),\ v)) = g(g(g(B,x),g(y,z)),g(u,v))$ and $g(f(g(g(N,x),y)),g(g(g(N,x),y),f(g(g(\ N,x),y)))) \neq g(g(g(x,f(g(g(N,x),\ y))),\ y),f(g(g(N,x),y)))$. Finally, 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

Table 6: Memory Requirement [KBytes]

Indexed Set	Standard Path–Indexing	Extended Path–Indexing 1	2	3	4
EC–pos	115	184	281	345	371
EC–neg	639	951	1759	2754	3822
CL–pos	437	570	887	1341	1885
CL–neg	1400	1908	3395	6223	10974
BOOL–pos	576	821	1550	1985	1985
BOOL–neg	1125	1498	2710	4047	5151

Table 7: Filter Properties [Terms Found]

Indexed Set	Query Set	No Filter	Standard Path–Indexing	Extended Path–Indexing 1	2	3	4	Perfect Filter
EC–pos	EC–neg	250000	249104	88791	52208	50434	50421	0
EC–neg	EC–pos	250000	249104	88791	52208	50434	50421	0
CL–pos	CL–neg	1000000	154399	118319	13684	3206	385	0
CL–neg	CL–pos	1000000	154399	118319	13684	3206	385	0
BOOL–pos	BOOL–neg	36000000	40773	20322	408	0	0	0
BOOL–neg	BOOL–pos	36000000	40773	20322	408	0	0	0

and BOOL–neg are $Sum(x, p(x, y), p(x,\ s(x,\ y)))$ and $\neg Sum(p(c2, n(x)), p(c2,\ x), c4)$.

Memory Requirements. In Table 6 we compare the memory requirements of Standard Path–Indexing and extended Path–Indexing. Extended Path–Indexing occupies the more memory the larger the NU–depth is. The NU–depth should be selected carefully, as the memory requirements may drastically increase. In most cases a NU–depth of 1 or 2 is sufficient, as our retrieval times in Table 8 will show.

Filter Properties. Table 7 gives a survey on how many terms pass the indexing filter. The column "No Filter" shows how many term pairs would have to be unified without using a filter. In our examples we created an index for the "Indexed Set" and started a query for each term of the "Query Set". Obviously, Standard Path–Indexing is a bad filter for the term sets EC–pos and EC–neg. The filter properties for CL–pos and CL–neg are average. Standard Path–Indexing is a very good filter for the term sets BOOL–pos and BOOL–neg. A "Perfect Filter" would not find any unifiable term pairs and therefore our examples are rather extreme. For all indexing problems Extended Path–Indexing significantly improves the filter properties. The larger the NU–depth the better the filter.

Retrieval Times. The experiments were run on a Sun SPARCstation SLC computer with 16 MBytes of RAM. The size of the hash table was limited to 500, the index depth (introduced in [McC92]) was limited to 15. We give a survey on the retrieval times in Table 8. The times include the construction of the query trees as well as the time spent for test unifications.

In the EC and CL examples Extended Path–Indexing is faster than Standard Path–Indexing. Taking the memory requirements into consideration a NU–depth of 1 or 2 should be preferred. However, as Standard Path–Indexing is already a very good filter for the BOOL examples, the performance cannot be improved because of the overhead produced by Extended Path–Indexing (the terms are rather small and therefore the test unifications are easy).

Table 8: Retrieval Times [Seconds]

Indexed Set	Query Set	No Filter	Standard Path–Indexing	Extended Path–Indexing 1	2	3	4
EC–pos	EC–neg	41.5	67.8	41.6	36.0	38.2	42.1
EC–neg	EC–pos	44.7	57.5	34.6	29.0	28.6	28.9
CL–pos	CL–neg	107.5	43.5	38.8	11.9	13.7	20.7
CL–neg	CL–pos	111.2	46.1	41.4	17.2	19.7	24.3
BOOL–pos	BOOL–neg	301.5	22.8	26.3	35.0	39.1	40.2
BOOL–neg	BOOL–pos	299.8	5.9	6.6	7.6	8.5	8.9

Although they have no application in theorem proving unifications of the form X–pos X–pos and X-neg X–neg have also been tested. They produced similar results.

Implementation. Standard and Extended Path–Indexing are implemented in C and are available via anonymous ftp. They are as well as implementations of discrimination and abstraction trees part of "A Collection of Indexing Data Structures (ACID)" developed at MPI. In the future we will try to further improve ACID which is a library for efficient data structures and algorithms for theorem provers. Our implementations do not depend on term data structures and can very easily be embedded into other C software. Our Path–Indexing software allows parallel access to the same index and is able to delete and insert entries into the index when retrieval is interrupted. For more information send e-mail to *acid@mpi-sb.mpg.de*.

8 Conclusion

We presented an extention of Standard Path–Indexing which is able to drastically reduce retrieval time in the cases in which Standard Path–Indexing itself is not a good filter. However, the advantages of Standard Path–Indexing are preserved. In Extended Path–Indexing indirect clashes and occur–check failures are detected with the help of extended Path–Lists at the time the extended query tree is evaluated. Therefore less test unifications have to be performed. The memory requirements can be reduced by an appropriate choice of the NU–depth.

Acknowledgements

Thanks to Leo Bachmair, Hans Jürgen Ohlbach, and David Plaisted for their comments on earlier versions of this paper. We also like to thank William McCune for providing the term sets.

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.

[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.

[HN81] L.J. Henschen and S.A. Naqvi. An improved filter for literal indexing in resolution systems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pages 528–529, 1981.

[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.

[McC90] W. McCune. Otter 2.0. In *10th International Conference on Automated Deduction*, pages 663–664. Springer Verlag, 1990.

[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.

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

[WP84] M. Wise and D. Powers. Indexing prolog clauses via superimposed codewords and field encoded words. In *Proceedings of the IEEE Conference on Logic Programming*, pages 203–210, 1984.