Term Indexing for the LEO-II Prover

Frank Theiß[1]
Christoph Benzmüller[2]
[1]FR Informatik, Universität des Saarlandes, Saarbrücken, Germany
lime@ags.uni-sb.de
[2]Computer Laboratory, The University of Cambridge, UK
chris@ags.uni-sb.de

Abstract

We present a new term indexing approach which shall support efficient automated theorem proving in classical higher order logic. Key features of our indexing method are a shared representation of terms, the use of partial syntax trees to speedup logical computations and indexing of subterm occurrences. For the implementation of explicit substitutions, additional support is offered by indexing of bound variable occurrences. A preliminary evaluation of our approach shows some encouraging first results.

1 Introduction

Term indexing has become standard in first order theorem proving and is applied in all major systems in this domain [RV02, Sch02, WBH$^+$02]. An overview on first order term indexing is given in [RSV01] and [NHRV01] presents an evaluation of different techniques. Comparably few term indexing techniques have been developed and studied for higher order logic. An example is Pientka's work on higher order substitution tree indexing [Pie03].

In this paper we present a new approach to higher order term indexing developed for the higher order resolution prover LEO-II[1], the successor of LEO [BK98]. Our approach is motivated by work presented in [TSP06], which studies the application of indexing techniques for interfacing between theorem proving and computer algebra.

Pientka's approach is based on substitution tree indexing and relies on unification of linear higher order patterns. While higher order pattern unification is a comparatively high level operation, the approach we present here is based on coordinate and path indexing [Sti89] and thus relies on lower level operations, for example, operations on hashtables. Apart from indexing and retrieval of terms, we particularly want to speedup basic operations such as replacement of (sub-)terms and occurs checks.

[1]The LEO-II project at Cambridge University has just started (in October 2006). The project is funded by EPSRC under grant EP/D070511/1.

2 Terms in de Bruijn Notation

LEO (and its successor LEO-II under development) is based on Church's simple type theory, that is, a logic built on top of the simply typed λ-calculus [Chu40]. In contrast to LEO, our new term data structure for LEO-II uses de Bruijn [dB72] indices for the internal representation of bound variables. In this paper, de Bruijn indices have the form x_i, where x is a nameless dummy and i the actual index. Constants and free variables in LEO-II, called symbols in the remainder of this paper, have named representations. Due to Currying, applications have only one argument term in LEO-II.[2]

Terms in LEO-II are thus defined as follows:

- *Symbols* are either constant symbols (taken from an alphabet Σ) or (free, existential) variable symbols (taken from an alphabet $\mathcal{V}$). Every symbol is a term.
- *Bound variables*, represented by de Bruijn indices x_i for some index $i \in \{1, 2, \dots\}$, are terms.
- If s and t are terms, then the *application* $s@t$ is a term.
- If t is a term, then the *abstraction* $\lambda.t$ is a term.

For bound variables x_i, the de Bruijn index i denotes the distance between the variable and its binder in terms of scopes. Scopes are limited by occurrences of λ-binders, thus the index i is determined by the number of occurrences of λ-binders between the variable and its binder.

For instance, the term

$$\lambda a.\lambda b.(b = ((\lambda c.(cb))a))$$

translates to

$$\lambda\lambda.(x_0 = ((\lambda.(x_0 x_1))x_1))$$

in de Bruijn notion. While de Bruijn indices ease the treatment of α-conversion in the implementation, they are less intuitive. As it can be seen in the above example, different occurrences of the same bound variable may have different de Bruijn indices. This is the case here for b, which translates to both x_0 and x_1. Vice versa, different occurrences of the same de Bruijn index may refer to different λ-binders. This is the case for x_0, which relates to both the bound variable b (first occurrence of x_0) and the bound variable c (second occurrence of x_0). Similarly, x_1 is related to the bound variables b and the bound variable a.

[2] Alternative representations, for example, spine notation [CP97], offer at first sight shorter paths to term parts that are relevant for a number of operations. The difference is primarily the order in which the parts of a term can be accessed. In the case of the spine notation, for example, the head symbol of a term can be directly accessed. In our approach we try to offer these shortcuts by representing indexed terms in a graph structure. This allows to adopt additional ways of accessing (sub-)terms by introducing additional graph edges. For instance, the head symbol of each term and its position are indexed in our data structure.

The presentation of LEO-II's type system (simple types) is omitted here. With respect to LEO-II's term indexing, typing only provides an additional criterion for the distinction between terms, for example, different occurrences of the same de Bruijn index may have different types. Apart from this, typing has no further impact on the indexing mechanism. Overall correctness is ensured outside the index structure, that is, by indexing only terms in $\beta\eta$ normal form.

3 The Index

The indexing mechanism we describe here supports fast access to indexed terms and its subterms. The index will be used in LEO-II to store intermediate results (consisting of clauses, literals, and terms in literals) of LEO-II's resolution based proof procedure. The idea is that these intermediate results are always kept in $\beta\eta$ normal form (that is, η short and β normal; for example, the term $(\lambda.(\lambda.(x_1 x_0))))((\lambda.g x_0)c)$ has the $\beta\eta$ normal form (gc)). Hence, $\beta\eta$ normalisation is an invariant of our approach and we assume that we never insert non-normal terms to an index.

Key features of our indexing mechanism are a shared representation of terms (see Section 3.1), the use of partial syntax trees to speedup logical computations (see Section 3.2) and the indexing of subterm occurrences (see Section 3.3). For the implementation of explicit substitutions [FKP96, ACCL90], additional support is offered by indexing of bound variable occurrences (see Section 3.4).

Partial syntax trees are used to index occurrences of symbols and subterms within a term. They help to avoid occurs checks and to early prune superfluous branches in the implementation of operations like replacement, substitution or β-reduction. Checking whether or not a symbol occurs within a term or in a given branch of its syntax tree requires only constant time.

The implementation of the index is furthermore based on the use of *cascaded hashtables*, for example, to index application terms according to their function or argument term. This allows requests for terms in a style similar to SQL [Ame92]. For example, indexing of applications is realised similar to an SQL table `Applications` featuring the columns `appl` for application terms, `func` for their respective function terms and `arg` for their argument terms. Retrieval of terms is similar to SQL queries like "`select appl from Applications where func=t`", which returns terms whose function term is `t`.

3.1 Shared Representation of Terms

Terms in LEO-II have a perfectly shared representation, that is, all occurrences of syntactically equal terms (in de Bruijn notation) are represented by a single instance. An exception are bound variables, where instances of the same variable may have different de Bruijn indices. The treatment of bound variables is described further in Section 3.4.

Terms are represented as *term nodes*. Term nodes are numbered by $n \in \{1, 2, \ldots\}$ in the order they are created. In the following, term nodes are referred to either by their number or by their graph representation, which is defined as follows:

- For each symbol $s \in \Sigma$ occurring in some term, a term node $\texttt{symbol}(s)$ is created.
- For each bound variable x_i occurring in some term, a term node $\texttt{bound}(i)$ is created.

- If an application $s@t$ occurs in some term, where s is represented by term node i and t by term node j, a term node $\texttt{application}(i, j)$ is created.

- If an abstraction λt occurs in some term, where t is represented by term node i, a term node $\texttt{abstraction}(i)$ is created.

This graph representation of terms is implemented using hashtables:

- Hashtable `abstr_with_scope` : $I\!N \rightarrow I\!N$ is used to lookup abstractions with a given scope i.

- Hashtable `appl_with_func` : $I\!N \rightarrow I\!N \rightarrow I\!N$ is used to lookup an application with a given function i and argument j.

- Hashtable `appl_with_arg` : $I\!N \rightarrow I\!N \rightarrow I\!N$ is used to lookup an application with a given argument j and function j. This is similar to `appl_with_func`, but the hashtable keys are used here in reversed order.

This hashtable system can be employed to retrieve term nodes in a similar way as in a relational database. It can be used to retrieve single terms as well as sets of terms, for example, all application term nodes whose function term is represented by node i.

3.2 Partial Syntax Trees

Term indexing in LEO-II is based on partial syntax trees (PST), a concept that is newly introduced in this paper. Partial syntax trees are used to indicate positions of a symbol or a particular subterm within a term. PSTs are called *partial* because they only represent *relevant* parts of a term. Examples are PSTs recording symbol occurrences in a term, where relevant part means, that the term part in question actually contains an occurrence of that symbol. Such PSTs allow for early detection of branches in a term's syntax tree with no occurrences of a specific symbol, since these branches are not represented in the PST for this symbol.

In LEO-II's term system (remember that this is based on simply typed λ-calculus with Currying) a term position is defined as follows:

- While symbol nodes and bound variable nodes have no children in a term's syntax tree, abstraction nodes respectively application nodes have exactly one child respectively exactly two children. The *relative position* of these children to their parent node is described by either `abstr` (the relation between an abstraction node and its scope), `func` or `arg` (the relation between an application node and its function term respectively its argument term).

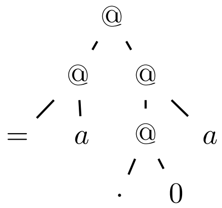
- A *position* is defined as a (possibly empty) sequence of *relative positions*. Starting from the top position in a term, each entry of the sequence describes one traversal step in the term's syntax tree.

- An empty sequence of *relative positions* represents the *root position* or *empty position*, which is the topmost position in a term.

Consider, for example, the term $(\lambda.x_0)@(f@a)$. Its subterms occur at the following positions:

$(\lambda.x_0)@(f@a)$	:	`[]`
$\lambda.x_0$	:	`[func]`
x_0	:	`[func; arg]`
$f@a$	:	`[arg]`
f	:	`[arg; func]`
a	:	`[arg; arg]`

Based on this notion of positions, we introduce the notion of partial syntax trees. As an example[3], consider the term $a = 0 \cdot a$, which translates in Curried form to $(= @a)@((\cdot@0)@a)$. The example term's syntax tree is given by:



A partial syntax tree (PST) is a tree of nodes corresponding to positions in a term. Each term position which occurs in a PST is represented as a node which

- has up to three child trees[4] (these children are partial syntax trees which correspond to the terms at one of the *relative positions* `abstr`, `func` or `arg`), and
- may be annotated by some data.

A partial syntax tree t is denoted by $pst(t_{abstr}, t_{func}, t_{arg})$, where t_{abstr} is the PST of the scope of t if t is an abstraction, and where t_{func} and t_{arg} are the PSTs of the function term and the argument term of t if t is an application. If no position in a branch of the syntax tree is annotated by some data, this branch is *empty* and is denoted by an underscore (_).

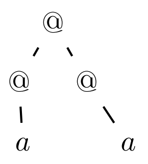
The PST corresponding to the whole term in the above example and its annotations is thus given by:

[3] We present a simple first order example here, since the the treatment of bound variables is special and is described later in Section 3.4.

[4] The data structure of PSTs can not only be *partial*, its structure can also *exceed* the syntax tree of terms as defined in Section 2 if all three children of a node are nonempty. This may be the case when using PSTs to represent *coordinates*, that is term positions which occur in *some* term. This is used when building the index as described in Section 3.3, which is similar to Stickel's path indexing and coordinate indexing methods [Sti89, McC92].

$$
\begin{array}{ll}
p_1 = pst(\_, p_2, p_5) & \\
p_2 = pst(\_, p_3, p_4) & \\
p_3 = pst(\_, \_, \_) & \text{with annotation } = \\
p_4 = pst(\_, \_, \_) & \text{with annotation } a \\
p_5 = pst(\_, p_6, p_9) & \\
p_6 = pst(\_, p_7, p_8) & \\
p_7 = pst(\_, \_, \_) & \text{with annotation } \cdot \\
p_8 = pst(\_, \_, \_) & \text{with annotation } 0 \\
p_9 = pst(\_, \_, \_) & \text{with annotation } a
\end{array}
$$

When the term is added to the index, however, not the PST of the entire term is recorded, but the PSTs of each of the occurring symbols (and subterms). For example, the PST of all occurrences of the symbol a in $a = 0 \cdot a$ is given by:

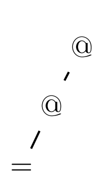


PST for a

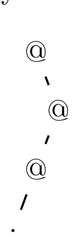
If a symbol occurs at a term position, the corresponding PST entry is annotated by that symbol. If a branch of a term's syntax tree has no occurrences of the symbol in question, the PST contains no entry for this branch. The PST for occurrences of symbol a in the above example is thus given by:

$$
\begin{array}{ll}
p_{a1} = pst(\_, p_{a2}, p_{a4}) & \\
p_{a2} = pst(\_, \_, p_{a3}) & \\
p_{a3} = pst(\_, \_, \_) & \text{with annotation } a \\
p_{a4} = pst(\_, \_, p_{a5}) & \\
p_{a5} = pst(\_, \_, \_) & \text{with annotation } a
\end{array}
$$

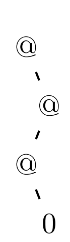
Similarly, the PSTs for the remaining symbols are recorded:



PST for $=$



PST for $\cdot$



PST for 0

If the PST of all occurrences of a symbol (or subterm) t' in a given term t is available, this provides a basis for speeding up replacements of t'. Also a costly occurs check is avoided, since the existence or non-existence of a PST for a symbol can be used as criterion. The nodes of the PST for t' determine the nodes in t that have to be modified when performing the replacement operation, and all nodes in t that are not represented in the PST for t' remain unchanged (i.e., the recursion over the term structure for replacement operations is pruned early).

When replacing a by $(f@b)$ in the above example, the operation proceeds as follows:

- The operations starts at root position with term $(= @a)@((\cdot@0)@a)$ and with the corresponding PST for a, $p_{a1} = pst(\_, p_{a2}, p_{a4})$. As both the function child p_{a2} and the argument child p_{a3} of the PST are nonempty, the replacement operation recurses over both the function term $(= @a)$ and the argument term $((\cdot@0)@a)$:

$$[(f@b)/a](= @a)@((\cdot@0)@a) \Rightarrow ([(f@b)/a](= @a))@([(f@b)/a]((\cdot@0)@a))$$

- To replace a in $(= @a)$ with corresponding PST $p_{a2} = pst(\_, \_, p_{a3})$, only the argument term has to be processed. The child PST corresponding to the function term in p_{a2} is empty, indicating that there are no further occurrences of a in this term. Thus we have:

$$[(f@b)/a](= @a) \Rightarrow (= @([(f@b)/a]a))$$

and analogously for $((\cdot@0)@a))$ and $p_{a4} = pst(\_, \_, p_{a5})$, where again processing the function term $(\cdot@0)$ is avoided:

$$[(f@b)/a]((\cdot@0)@a)) \Rightarrow ((\cdot@0)@([(f@b)/a]a))$$

- Finally a is replaced in the term a with corresponding PST p_{a3} respectively p_{a5}. Both p_{a3} and p_{a5} have no child nodes and are annotated with a, so the result is in both cases the replacement term $(f@b)$:

$$[(f@b)/a]a \Rightarrow (f@b)$$

The result of this operation is thus:

$$[(f@b)/a](= @a)@((\cdot@0)@a) \Rightarrow (= @(f@b))@((\cdot@0)@(f@b))$$

During the operation, only those branches of the syntax tree with an actual occurrence of a are processed and branches with no occurrences of a, here the terms $=$ and $(\cdot@0)$, are avoided. In this example, only five out of nine term nodes have to be processed due to the guidance provided by the PST.

As a probably useful indicator for the speedup for replacements obtainable this way we therefore investigate the ratio of term size to PST size counted in nodes of the tree, that is, the number of abstractions, applications, symbols and bound variables. As is illustrated above, this ratio is a measure for the speedup which we expect for replacement operations.

In the above example, the term size is 9 (we have 9 nodes), which gives the following rates for the occurring symbols:

Symbol	PST size	PST/term size
a	5	0.56
$=$	3	0.33
$\cdot$	4	0.44
0	4	0.44

We examined an excerpt of Jutting's Automath encoding of Landau's book Grundlagen der Analysis [vBJ77, Lan30] with over 900 definitions and theorems (see Section 4 for details) and found an average PST size/term size rate of 0.21 for symbol occurrences. When indexing nonprimitive terms, too (that is, applications and abstractions), this rate dropped to 0.12.

3.3 Building the Index

The index records whether and at which positions a subterm[5] occurs in a term. Similar to relational databases, both subterms occurring in a given term and terms in which a given subterm occurs are indexed. Thus, the index can be used to find terms in the database with occurrences of particular subterms and also to speed up logical operations such as substitution by avoiding occurs checks.

The index is built recursively, starting from symbols, which are the leaf nodes in a term's syntax tree. The only term which occurs in a symbol is the symbol itself at root position. Nonprimitive terms, that is abstractions and applications are built up as follows:

- The subterms occurring in an *abstraction* are all subterms which occur in the scope of the abstraction. For a symbol whose occurrences in a term A are recorded in the PST t', its occurrences in the abstraction $\lambda.A$ are given by $t = pst(t', \_, \_)$.
- The subterms occurring in an *application* are all subterms which occur in its function term or in its argument term. For a symbol whose occurrences in the function and argument term are recorded in the PSTs t'_{func} and t'_{arg}, the PST t recording its occurrences in the application is given by $t = pst(\_, t'_{func}, t'_{arg})$. If the term occurs only in the argument respectively in the function of an application, t_{func} respectively t_{arg} is *empty*.

Furthermore each primitive and nonprimitive term is recorded to occur as a subterm of itself at root position.

The result is a PST for each subterm of the term to be indexed, describing the occurrences of this subterm. These PSTs are added to the hashtable `occurrences`. Additionally, terms are indexed according to their subterms in a second hashtable `occurs_in`. A third hashtable is `occurrs_at`, which is used to index terms according to subterms at a given term position. Thus the core of the index consists of:

[5] In the current implementation, both occurring symbols and nonprimitive subterms are indexed. However, we plan to further evaluate the tradeoff between the speedup gained this way and the cost for maintenance of the index. Depending on this evaulation, we may want to restrict the nonprimitive subterms to be indexed, for example, by using their size as a criterion. Similar ideas have been examined by McCune [McC92], for example, the effect of limitations on the length of paths used in path indexing.

- Hashtable `occurrences` : $\mathbb{N} \to \mathbb{N} \to PST$ indexes occurrences of subterms (the second key) in a given term (the first key). The indexed value is a PST of the positions where the subterm occurs. If a subterm does not occur, then there is no entry in the hashtable.
- Hashtable `occurs_in` : $\mathbb{N} \to \mathbb{N}^*$ is used to index a list of all terms in which a given subterm (the key) occurs.
- Hashtable `occurrs_at` : $pos \to \mathbb{N} \to \mathbb{N}^*$ is a hashtable to index all terms in which a given subterm (the second key) occurs at a given position (the first key).

For example, occurrences of symbol a in the example term $(= @a)@((\cdot @0)@a)$ are indexed by the following hashtable updates (we assume that a is represented by term node i and $(= @a)@((\cdot @0)@a)$ by term node j):

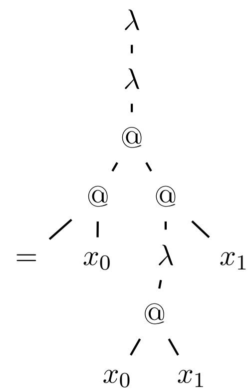
- add pst_a with first key j and second key i in `occurrences`
- add j with key i in `occurs_in`
- add j to the set hashed in `occurs_at` with first key [`func`; `arg`] and second key i; if no such set exists in the hashtable, add the singleton $\{j\}$
- add j to the set hashed in `occurs_at` with first key [`arg`; `arg`] and second key i; if no such set exists in the hashtable, add the singleton $\{j\}$

The basic operations of adding a term to the index take constant time (except for rehashing). The indexing of a term of length n takes time $O(n)$.

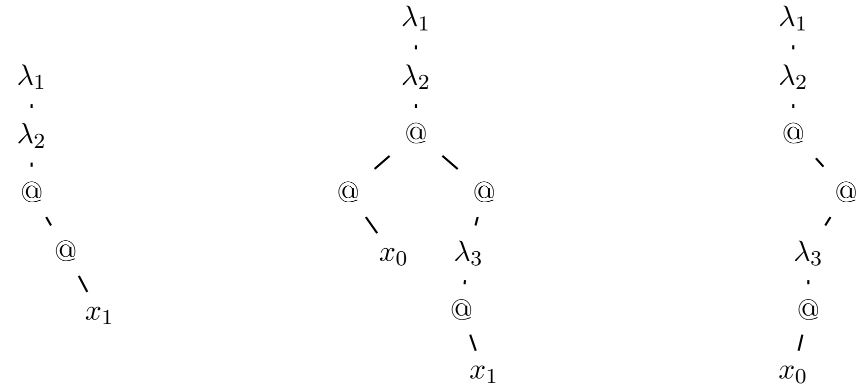
3.4 Bound Variables

Bound variables play a special role in the term system. To see this, remember our example from the beginning, that is, the term $\lambda a.\lambda b.((= b)((\lambda c.(cb))a))$ or, with de Bruijn indices, $\lambda\lambda.((= x_0)((\lambda.(x_0 x_1))x_1))$. This example shows that two occurrences of the same bound variable may have syntactically different de Bruijn indices and that the de Bruijn indices of occurrences of different variables may be syntactically equal. It is desirable to provide quick access to all variables bound by a given binder to speedup β-reduction and related operations such as raising or lowering of bound variable indices. We will now illustrate our solution to this issue. Remember that indexed terms are always kept in $\beta\eta$ normal form, hence, normalisation is mandatory after instantiation of existential variables or expansion of defined constants (if the modified terms shall be indexed again).

The syntax tree of our example term in de Bruijn notation is



In this case the bound variable b has two instances which are denoted by x_0 and x_1, while x_0 (resp. x_1) can denote both c or b (resp. b or a). Since bound variables are indexed as described in Section 3.3, this gives a somewhat scattered information on where to find the variables that are bound by one particular λ-binder. This kind of information, however, is important in practice, for example, to support efficient β-reduction. We therefore once more employ PSTs to describe the occurrences of variables bound by one and the same λ-binder:



Variables bound by λ_1 Variables bound by λ_2 Variables bound by λ_3

For example, the PST indicating occurrences of variables bound by λ_2 in the above example is given by:

$$
\begin{array}{ll}
p_1 = pst(p_2, \_, \_) & \\
p_2 = pst(p_3, \_, \_) & \\
p_3 = pst(\_, p_4, p_6) & \\
p_4 = pst(\_, \_, p_5) & \\
p_5 = pst(\_, \_, \_) & \text{with annotation 0} \\
p_6 = pst(\_, p_7, \_) & \\
p_7 = pst(p_8, \_, \_) & \\
p_8 = pst(\_, \_, p_9) & \\
p_9 = pst(\_, \_, \_) & \text{with annotation 1}
\end{array}
$$

The explicit notation of variables bound by λ_2 and λ_3 is analogous and therefore omitted here.

This list of PSTs is also recorded in LEO-II's index, in the order shown above. Each PST is assigned a *scope number*, where the scopes are defined by the occurrence of λ-binders. When traversing the syntax tree, the PST recording occurrences of variables bound by the *first* λ-binder is assigned scope number 1, the PST related to the *second* λ-binder has scope number 2 and so on.

While the term $\lambda((\lambda.x_0) = ((\lambda.(x_0x_1))x_1))$ is closed, that is, all de Bruijn indexed variables are bound by a λ-binder within the term, this is not always true for its subterms. Unbound variables occur, for example, in $\lambda.(x_0x_1)$, where x_1 refers to a binder outside the term. In particular, all primitive terms consisting only of de Bruijn variables refer to a binder outside this term. While the PSTs for bound variables can be constructed as shown above, the determination of the scope numbers deserves a special treatment in case of *loose bound variables*, that is bound variables without a binder in the given subterm. If a term has occurrences of loose bound variables, their de Bruijn index allows to determine the distance to their (virtual) binder measured in scopes upwards from the term's root position. PSTs for loose bound variables are assigned a scope number $s \leq 0$. The PST to denote all occurrences of x_1 in itself is consequently assigned the scope number -1, and the PST denoting the occurrence of x_1 in $\lambda.(x_0x_1)$ is assigned scope number 0.

Indexing of bound variable occurrences in a term is used to speedup β-reduction. For each λ-binder the positions of variables bound by this binder are known, thus, only the parts of the term that actually are modified have to be processed. In the context of explicit substitutions [FKP96, ACCL90], the implementation of shift and lift operators can furthermore be reduced to recalculation of the offset and elimination of bound variable PSTs from the list.

3.5 Using the Index

3.5.1 Adding and Retrieving Terms:

Terms are added to and retrieved from an index in a similar way as in coordinate or path indexing [Sti89]. When a term t is added to the index, the PSTs of symbol occurrences are constructed as described in Section 3.3. Then the following hashtables are updated:

- In `occurrences`, the PSTs of the occurring symbols (or subterms) are added.
- For each occurring symbol (or subterm), t is added to the set of terms which is recorded for that symbol in `occurs_in`. If there is no such entry in `occurs_in`, the singleton $\{t\}$ is added.
- For each term position in t, t is indexed in `occurs_at` in the same way with the position as first key and the the subterm as second key.

Furthermore, the PSTs for bound variables are constructed as described in Section 3.4 and are added to the hashtable `boundvars`.

For each occurrence of a subterm at a given position in a query term, a set of candidate terms is retrieved from hashtable `occurs_at`. Sets of candidate terms can furthermore be retrieved from hashtable `occurrences` for subterms occurring at unspecified positions. The result of the query is the intersection of all candidate sets obtained this way.

3.5.2 Speeding up Computation:

Using the index, efficient occurs checks are reduced to single hashtable lookups. Efficient replacement of a symbol or subterm t is furthermore supported by PSTs recorded in the index (in hashtable `occurrences`), since these PSTs make it possible to avoid processing of term parts with no occurrences of t. Fast β-reduction is supported by the PSTs recorded in hashtable `boundvars`.

3.5.3 Explicit Substitutions:

Our approach can also support explicit substitutions. Note that subterm occurrences in a term t can be quickly determined as described above. Similarly, the occurrences of a subterm s in the result of applying a substitution σ to a term t can be determined using our indexing technique. To determine occurrences of s in σt where $\sigma = [b/a]$, occurrences of s and a in t are looked up from the index, as well as occurrences of s in b. Thus we get three PSTs $pst_{s/t}$, $pst_{a/t}$ and $pst_{s/b}$. To find all occurrences of s in σt, all positions annotated by a in $pst_{a/t}$ are replaced by a new sub PST $pst_{s/b}$, and the result is merged with $pst_{s/t}$. For $\sigma_1\sigma_2\dots\sigma_n t$, this operation is cascaded.

4 Preliminary Evaluation

Full evaluation of the indexing method presented here is still work in progress. This is because the implementation of LEO-II is still at its very beginning, so that we cannot pursue an empirical evaluation within theorem proving applications with LEO-II at the current stage of development. A purely theoretical examination is difficult and furthermore questionable, as the computational complexity can be expected to heavily depend on the structure of the application domains [NHRV01].

However, we were able to undertake some first experiments which may give us an impression of the efficiency gain we may expect for LEO-II (for example, in comparison to LEO and other higher order theorem provers that do not use term indexing techniques).

In order to get a realistic impression of the structural characteristics of real world term sets, we indexed a sample selection of 900 theorems and definitions from a HOTPTP [GS06] version of Jutting's Automath encoding of Landau's book Grundlagen der Analysis [vBJ77, Lan30]. An overview on the results of this experiment is given in Figure 1. We will now discuss these results.

In our study, we determined, for example, the rate of term sharing, which is the average number of parent nodes per node and the average number of terms a given node occurs in. At first sight the average number of parent nodes of 1.68 appears to be relatively low, an impression which is underlined by the high number of nodes with no or one parent node (about 90%). For nodes which are deeply buried in a term's structure, however, the sharing rate multiplies along the path up to root position, so the average number of terms a node occurs in (33.5) relativises this impression. Our experiments indicate furthermore an increase of the sharing rate by operations like

Number of indexed terms	977
Number of created term nodes	11618
Average term size	54
Number of nodes with no parent nodes	904
Number of nodes with one parent node	9633
Number of nodes with two more more parent nodes	1083
Maximum number of parent nodes	2778 (symbol $\forall$)
Average number of parent nodes	1.68
Average number of terms a node occurs in	33.5
-”-(for symbols)	493.9
-”-(for nonprimitive term nodes)	24
Average PST/term size for symbol occurrences	0.21
Average PST/term size for bound variable occurrences	0.33
Average PST/term size for all term nodes	0.12

Figure 1: Structure of the Landau sample.

replacement, substitution and β-reduction, due to the reuse of already indexed subterms. Additionally, the maintenance of the index is supported by data already existing in the index. As most logical operations on terms reuse parts of these terms, the cost to maintain the index is less than indexing a set of terms starting from an empty index, as required for instance, when initially loading a mathematical theory to memory.

An indicator for the term retrieval performance is the average number of terms a node occurs in. With an average number of occurrences of 33.5 and a total of 11618 term nodes, a theoretical average of 99.7% of candidate nodes for retrieval can be excluded by checking occurrences of subterms only (compared to a naive approach). By specifying the position of the subterm's occurrence, the set of retrieved terms is further restricted.

The use of shared terms is responsible for a further improvement of performance similar to the transition from coordinate indexing to path indexing [Sti89]. While both methods employ a common underlying idea, path indexing is substantially faster. In the former approach terms are discriminated by occurrences of single symbols at specified positions (or *coordinates*, hence the name). The criterion in the latter is the occurrence of a *path*, that is the occurrence of a sequence of specified symbols in a descending path in the syntax tree. The retrieval of candidates for one path of length n is thus corresponding to n passes of retrieval in coordinate indexing. We expect a similar effect in our approach due to shared representation of terms, since terms are indexed according to the occurrence of nonprimitive term structures, too. This assumption is supported by the increase of the exclusion rate of 95.7% for symbol occurrences only (with an average number of 493.9 superterms per node) to 99.8% for nonprimitive terms (with an average of 24 superterms per node). This rise corresponds to a theoretical speedup by factor 20.

We can also predict a significant performance improvement of operations, such as replacement, substitution and occurs check. They all are critical in theorem proving. The indexing method we present here supports occurs checks in constant time, based on simple hashtable lookup. This applies not only to symbols, but also to nonprimitive terms. This also supports global replacement of defined terms (for example, $a = b$) by their definiendum (for example, $\forall P.Pa \Rightarrow Pb$).

A measure of the efficiency improvement for replacement operations is the PST/term size rate. The value is 0.21 for symbols, which is relevant, for example, for variable substitution, and which corresponds to a theoretical speedup by factor 5. The value for bound variables, which is relevant for β-reduction, is 0.33, corresponding to a theoretical speedup by factor 3. The probably least common operation is the replacement of nonprimitive terms, as discussed above.

We are aware of the fact that the results shown here are based on a theoretical juggling with average values. These results may thus differ strongly from the behaviour when used in a realistic application in theorem proving as we intend. This is due to several factors: First we expect the structure of the set of indexed terms to change during operation. In general the basic operations of a theorem prover will increase the sharing rate of some symbols and subterms. This makes occurrences of these terms a less discriminating criterion, on the other hands it decreases the cost of maintenance of the index. The tradeoff of these two factors will be subject to further examination. Second, the evaluation of average values does most likely not correspond to index operation sequences as they actually occur in a realistic theorem proving application.

5 Conclusion and Future Work

The main features of the new higher order term indexing method we presented in this paper are shared term representation, relational indexing of subterm occurrences and the use of partial syntax trees. Occurrences of subterms are indexed in several ways and can be flexibly combined to design customised procedures for term retrieval and heuristics. Our method furthermore provides support for potentially costly operations such as global unfolding of defined terms. Our indexing method is based on simple hashtable operations, so there is little computational overhead in term retrieval and maintenance of the index. Indexing of subterm occurrences allows furthermore for an occurs check in constant time. Additionally, the performance is improved by the use of PSTs. Finally, a shared representation of terms helps to keep the costs for maintaining the index low and improves the performance of retrieval operations.

The indexing technique presented in this paper has been implemented in OCaml [LDG+05]. A proper evaluation of the approach within a real theorem proving context is still work in progress. However, first experiments are promising.

The preliminary evaluation in this paper is based on some statistical data we computed for 900 example terms from an encoding of Landau's textbook. To what extend our predictions on efficiency gain are realistic will be examined in future work.

Furthermore, as the experience from first order term indexing shows, most successful systems employ a combination of various indexing methods which are used complementarily. We will thus also evaluate which aspects of our indexing method result in a real performance gain and which do not. Our evaluation will be done with the LEO-II prover as soon as a first version of its resolution loop is available. In the LEO-II context we are particularly interested in the fast determination of clauses (resp. literals and terms in clauses) with respect to certain filter criteria. In the extreme case, these criteria may be based on complex operations such as higher order pattern unification [PP03] or even full higher order unification [Hue75, SG89].

Our approach differs from Pientka's work, which has a stronger emphasis on term retrieval. Pientka's method is based on high level operations such as unification of linear higher order patterns to construct substitution trees, while our method relies mainly on simpler low level operations and makes strong use of hashtables. Both methods appear complementary to some extend, which motivates the study of a combination of both.

Future work also includes the investigation of alternative term representation techniques, such as suspension calculus [Nad02], spine representation [CP97] and explicit substitutions [FKP96, ACCL90] in the context of our term indexing approach. We are especially interested in the combination of aspects from different representation techniques within a single graph structure.

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[Ame92] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. American National Standards Institute, pub-ANSI:adr, 1992. Revision and consolidation of ANSI X3.135-1989 and ANSI X3.168-1989, Approved October 3, 1989.

[BK98] Christoph Benzmüller and Michael Kohlhase. LEO – a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, number 1421 in LNAI, pages 139–143, Lindau, Germany, 1998. Springer.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Pittsburgh, PA, 1997.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[FKP96] Maria C. F. Ferreira, Delia Kesner, and Laurence Puel. Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization. In *Algebraic and Logic Programming*, pages 284–298, 1996.

[GS06] Allen Van Gelder and Geoff Sutcliffe. Extending the tptp language to higher-order logic with automated parser generation. In *Proceedings of IJCAR*, 2006.

[Hue75] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Lan30] Edmund Yehezkel Landau. *Grundlagen der Analysis*. Erstveröff, Leipzig, first edition, 1930.

[LDG+05] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Remy, and Jerome Vouillon. *The Objective Caml system, release 3.09*, October 2005.

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

[Nad02] G. Nadathur. The suspension notation for lambda terms and its use in metalanguage implementations, 2002.

[NHRV01] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. On the evaluation of indexing techniques for theorem proving. In *Proceedings of IJCAR 2001*, volume 2083 of *LNAI*, pages 257–271. Springer Verlag, 2001.

[Pie03] Brigitte Pientka. Higher-order substitution tree indexing. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2003.

[PP03] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification, 2003.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AICOM*, 15(2-3):91–110, jun 2002.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[SG89] W. Snyder and J. Gallier. Higherorder unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(2):101–114, 1989.

[Sti89] M. Stickel. The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.

[TSP06] Frank Theiß, Volker Sorge, and Martin Pollet. Interfacing to computer algebra via term indexing. In *Proceedings of Calculemus*. Elsevier, 2006.

[vBJ77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system*. PhD thesis, Technische Hogeschool Eindhoven, Stichting Mathematisch Centrum, 1977. See also: [Lan30].

[WBH+02] Ch. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, Ch. Theobald, and D. Topic. SPASS version 2.0. In *Proceedings of CADE 2002*, pages 275–279, 2002.