Interfacing to Computer Algebra via Term Indexing

Frank Theiß[1], Volker Sorge[2] and Martin Pollet[1]

[1] *Universität des Saarlandes, Germany*
`{lime,pollet}@ags.uni-sb.de`
[2] *School of Computer Science, University of Birmingham, UK*
`v.sorge@cs.bham.ac.uk`

Abstract

Computer Algebra Systems provide large collections of algorithms, which can be exploited to perform simplifications in Automated Theorem Proving systems. However, determining applicable algorithms and suitable instantiations of arguments in a dynamically changing proof situation is a potentially costly operation in particular in large proofs, as it requires matching the abstract description of an algorithm against actual terms in the current proof. We describe a new indexing method to handle the computational complexity of this task. It is based on a variant of coordinate indexing with shared term representations and operates as an active term filter, where term matching is triggered by the insertion of new terms into the index. An interface to Computer Algebra Systems can be realised by compiling a set of abstract descriptions of an algorithm into a term index structure composed of a static set of query terms. This enables a reactive behaviour of the indexing by automatically determining what simplification algorithms can be applied to newly occurring terms in a proof once they are inserted into the index structure.

Key words: Term Indexing, Interface CAS-ATP

1 Introduction

Making the computational power of Computer Algebra Systems (CAS) available in deduction systems has been the basis for much research in the last decade. One of the problems of an integration of CAS into deduction systems is to identify applicable algorithms in a given proof situation and to provide suitable arguments to invoke them. Since most of the research so far was concerned with the correct integration of CAS into mainly interactive deduction systems [1,3,11,13], applicability conditions of CAS algorithms could be encoded into inference rules, along with the extraction of their parameters from actual formulae and the reintegration of their results into the proof. Whether and where an algorithm is applicable is thus determined by matching the specification of an inference rule against terms in the current proof state. While this is still feasible in interactive theorem proving, when attempting to automate simplifications performed by CAS in the context of fully Automated Theorem Proving (ATP), an exhaustive search for applicable algorithms, by matching every proof node against the specifications of every algorithm, is computationally too expensive, in particular in large proofs with many possibly applicable algorithms.

with information on a set of abstract descriptions, or query terms, for CAS algorithms. This is achieved by compiling the set of query terms into a net of *meta nodes* and relate them to the term indexing structure. Thus once a new term is constructed during proof search, its insertion into the term indexing graph automatically yields a set of applicable CAS algorithms, which can be executed (we describe this procedure in Section 3).

There exists a variety of indexing techniques in particular in the context of first-order ATP [5], and the application of indexing techniques has become standard in all major systems like Spass [14], Vampire [8], or E-Prover [9]. In higher order systems, there are fewer examples, but indexing techniques are applied here, too. Isabelle provides discrimination nets as a filtering tool for fast selection of rules. Terms are classified by the symbol list obtained by flattening them in pre-order. Higher order aspects are simplified, e.g. λ-abstractions are regarded as unknowns [6]. In Twelf, higher order substitution tree indexing is used in higher order tabled logic programming and led in some examples to a speed up of over 800% [7]. All of these examples are based on substitution trees or discrimination trees and are, with the exception of Isabelle's discrimination nets, implemented in the context of machine-oriented proof methods and therefore focus on fast unification.

Our indexing method, on the other hand, is based on a variant of *coordinate indexing* [12]. Redundancies are minimised by the use of a shared representation of terms, which allows us — in contrast to the techniques discussed above — to search the index inside-out, that is, terms are identified according to occurrences of subterms at arbitrary positions. This feature is particularly important in our context as CAS algorithms are often applied in rewrite rules in an ATP, e.g. to simplify subterms of a formula, by recursively simplifying a term starting with the innermost subterm. The novelty of our approach lies in its reactive behaviour for the retrieval of possible simplifications for newly constructed terms. Our mechanism has been developed for the Ωmega system [10], a proof environment for the development of higher order proofs and is intended to be used fully automatically in Ωmega's proof planning facility. While we make use of Ωmega's higher order syntax and can handle higher order terms including λ-abstraction, we do not exploit advanced features such as higher order unification or matching.

2 Indexing Structure

Indexing is a method for fast retrieval of terms from a set of indexed terms or, conversely, to detect quickly that no such term in the index exists. For this the index classifies the terms according to some discriminating property. In our approach, this discriminating property is the occurrence of specified subterms at specified positions. The main index structure we use is that of a *positional tree* from Stickel's [12] approach of coordinate path indexing. In a positional tree nodes represent term positions and record both all terms in which the position occurs and the related subterm found at this position. Unlike the original approach of coordinate indexing we employ a further graph structure, the *term graph*, for a shared representation of indexed terms. Apart from reducing redundancies in the retrieval operation, the combination of positional tree and term graph allows us to access the index in two ways. On the one hand, all terms with an occurrence of a given sub term at a specified position can be looked up via the positional tree. On the other hand for a given subterm we can determine all terms containing that subterm at any position via the term graph. Since both structures are intertwined some parts of the exhibition of the term graph can only be fully appreciated after the description of the positional tree.

Before introducing the two concepts, we first introduce some basic notions necessary to describe our indexing mechanism. Given a set of constant symbols $\mathcal{C}$ and a set of variable symbols $\mathcal{V}$, we define the set $\mathcal{L}$ of terms of our higher order language as

- every element in $\mathcal{V}$ and $\mathcal{C}$ is in $\mathcal{L}$,
- if $f \in \mathcal{L}$ is an n-ary function symbol and $a_1, \ldots, a_n \in \mathcal{L}$ are terms then the *application* $f(t_1, \ldots, t_n)$ is in $\mathcal{L}$,
- if $r \in \mathcal{L}$ then the *abstraction* $\lambda.r$ is in $\mathcal{L}$.

We call constants and variables *primitive terms* to distinguish them from the *non-primitive terms* applications and λ-abstractions. Observe that $\mathcal{L}$ is intrinsically a higher order language. Indeed the elements of both $\mathcal{V}$ and $\mathcal{C}$ can be n-ary function symbols and are generally typed. For simplicity we omit types in the exposition in this paper. However, the presented index structures extend to typed terms as well.

Observe also that in the definition of λ-abstraction we use deBruijn indices [2]; that is, bound variables are not explicitly represented in the binder but are expressed by measuring the distance from the scope to a bound variable. As an example the expression $\lambda a.(\lambda b.f(a + b))a)$ is translated to $\lambda.(\lambda.f(x_1 + x_0)x_0)$ in anonymous notation. Bound variables in the direct scope of their λ-binder are named x_0 (i.e. deBruijn index 0), inside the next λ-binder the index used in the scope of the λ-abstraction is incremented by one, therefore the first occurrence of a in the above example translates to x_1, the second to x_0. Elements of $\mathcal{V}$ are restricted to variables of the form $x_i, i = 0, 1, \ldots$.

We use *query terms* to retrieve sets of *indexed terms* from our index structure. Query terms are partially specified terms that contain *query variables*, which serve as placeholders in order to match them against terms in the term index. We therefore extend our language $\mathcal{L}$ by a set $\mathcal{Q}$ of query variables. Elements of $\mathcal{Q}$ can be used similarly to constant elements in the construction of terms and we will denote them by small Greek letters. In addition we define a special type of query variable $\Phi(t)$. It serves as a placeholder for a term that

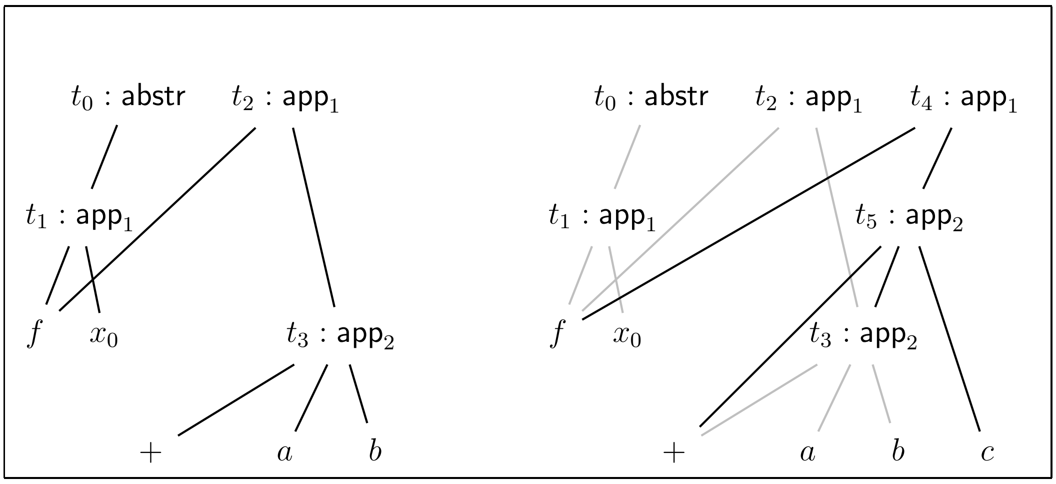

Fig. 1. The term graph on the left represents the terms $t_0 : \lambda.f(x_0)$ (in standard notation: $\lambda x.f(x)$) and $t_1 : f(a + b)$. The right side shows the graph after $t_4 : f((a + b) + c)$ has been inserted. Note that the subterm $t_3 : a + b$ is shared and thus not created when adding $t_4 : f((a + b) + c)$. Every symbol is represented by a single node.

contains some term t as a subterm.[1] We explain the role of both types of query variables more precisely in section 2.3.

2.1 Term Graph

We define the term graph $\mathsf{ID}_\mathcal{G}$ as a directed acyclic graph. Every node represents a term, where primitive terms are represented by leaf nodes and non-primitive terms by interior nodes of the graph. Each node is identified by a unique name that indicates the type of term it represents:

- A node representing a primitive term is named with $s \in \mathcal{V} \cup \mathcal{C}$. A node of this type does not have any children.
- A node representing an application of the form $f(a_1, \ldots, a_n)$ has a name of the form $t : \mathsf{app}_n$, where t is unique in $\mathsf{ID}_\mathcal{G}$ and app_n indicates that the term is an n-ary application. The node has $n + 1$ child nodes, one for the function and n for the argument terms.
- A node representing an abstraction is named $t : \mathsf{abstr}$, where t is unique in $\mathsf{ID}_\mathcal{G}$. This node has one child representing the term in the scope of the abstraction.

We call the nodes of an index graph also the *indexed terms*. When the distinction is not crucial we use the same identifier for terms and nodes. Two examples of term graphs can be seen in Figure 1.

[1] Note that Φ is not a higher order function but serves only as notational convenience. Note also that Φ can have more than one argument, but for the ease of explanation we omit this detail here.

Each node in the term graph $t \in \mathsf{ID}_{\mathcal{G}}$ has several attributes containing the necessary indexing information. Node attributes are denoted by the node name and the attribute name, separated by a dot. In general attributes named T indicate a link to the term graph, D point to the positional tree, and M is a link to meta nodes. The latter two will be explained in more detail in Sections 2.2 and 3, respectively. Concretely the attributes of a node $t \in \mathsf{ID}_{\mathcal{G}}$ are:

- $t.T_{sub}$ is the list of subterms of the term represented by t and contains all child nodes of t. If t is an application, $t.T_{sub}$ contains the terms representing the function and its argument. If t is of the form $\lambda.t'$ then $t.T_{sub} = \{t'\}$. $t.T_{sub}$ is empty if t is a primitive term.
- $t.T_{super}$ is the list of direct super-terms, that is, it contains all parent nodes of t.
- $t.D_{this}$ is the list of the nodes in the positional tree corresponding to t. $t.D_{this}$ records all positions t occurs in, in all other terms t' that are currently in the proof. See Section 2.2 for details.

Examples of term graphs are shown in Figure 1. The term graph on the left is produced from the insertion of the two terms $\lambda.f(x_0)$ and $f(a+b)$. The node t_2, for instance, represents a unary application and its list of subterms is $t_2.T_{sub} = \{f, t_3\}$ while $t_2.T_{super}$ is empty. The node f, on the other hand, represents a primitive term and thus has an empty set of subterms but has two direct super-terms, namely t_1 and t_2. After inserting the term $f((a+b)+c)$ we obtain the term graph on the right and the set of parent nodes of f is now of the form $f.T_{super} = \{t_1, t_2, t_4\}$.

When a new term t is added to the term graph, the shared representation is preserved; that is, if t is already in the index, then the corresponding graph node is returned. If t is not yet in the index, then the graph is extended. For non-primitive terms the components are inserted recursively. In detail the algorithm inserts a term t as follows (observe that we use t both for the term and the node representing that term):

(i) If t is a primitive term then return the corresponding node. If such a node does not exist, a new node for t with $t.T_{super} = \emptyset$ is introduced.
(ii) If t is of the form $\lambda.t'$, then insert t'. If $t : \mathsf{abstr} \in t'.T_{super}$ then return $t : \mathsf{abstr}$, otherwise create new node $t : \mathsf{abstr}$ with $t.T_{sub} = \{t'\}$ and insert t into $t'.T_{super}$.
(iii) If t is of the form $t_0(t_1, \ldots, t_n)$, then insert $t_i, i = 0, \ldots, n$. In case $\{t{:}\mathsf{app}_n\} = \bigcap_{i \in \{0,\ldots,n\}} t_i.T_{super}$ then return $t : \mathsf{app}_n$. Otherwise create new node $t : \mathsf{app}_n$ with $t.T_{sub} = \{t_0, \ldots, t_n\}$ and insert t into all $t_i.T_{super}$.

Whenever a new node has been introduced, all of its subterms are recorded in the respective position in the positional tree. This is explained in detail in Section 2.2.

As an example consider the term graph containing already $\lambda.f(x_0)$ and $f(a+b)$ in Figure 1. When inserting the term $f((a+b)+c)$ case iii of the

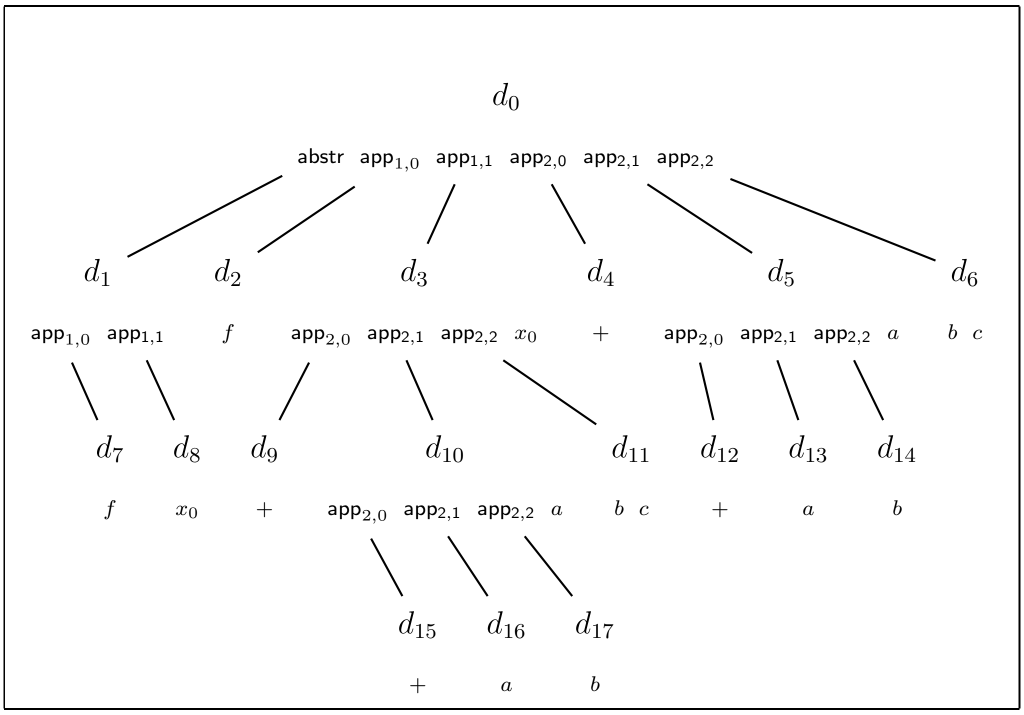

Fig. 2. Positional tree for the term graph of Figure 1.

insertion algorithm is executed and the subterms are recursively inserted. In order to insert the innermost non-primitive subterm $a + b$, first the primitive nodes $+$, a and b have to be inserted. Since they are all already contained in the tree no new nodes have to be introduced and the existing nodes are returned. As $t_3 : \mathsf{app}_2$ is a direct super-term of each of the subterms $+$, a and b, it is identified as the representation for $(a+b)$ and returned. The next term to be inserted is c, for which a new node has to be created. As there is no common super-term of $+$, t_3 and c, a new node t_5 is introduced for $(a+b)+c$, and t_5 is added to the list of super-terms of t_3. Similarly a new node t_4 is introduced for the whole term $f((a + b) + c)$.

Employing the shared term graph allows us to represent all occurrences of a term t represented by a single entity, namely a node in the graph. This allows terms to be enumerated and reduces equality checking of terms and operations on sets of terms to operations on natural numbers, which we use to label nodes in the practical implementation. While the term graph stores relations of subterms to each other, it does not yet contain information on the positions particular subterms can occur at in terms found in the proof state. This is done in the positional tree, which completes our indexing structure.

2.2 Positional Tree

The positional tree $\mathsf{ID}_{\mathcal{D}}$ is a labelled tree that represents all possible term positions that occur in a given proof state. The positions are stored as *relative positions* as edge labels. The nodes record all possible subterms that occur at the position given by the path from that node to the root in some existing term in the proof. An example of a positional tree is given in Figure 2, which is the corresponding tree for the term graph from Figure 1. Edge labels are depicted on top of the edges. In each node d of the tree all pairs (t, t') are recorded, where t' is the subterm of t at position d. While in Figure 2 the complete entries are omitted for clarity, positions with occurrences of the symbols a,b,c,f,$+$ and x_0 are marked (the full list of node entries is described below). If not explicitly distinguished we will use the same identifier for both the node and the position it represents.

More formally $\mathsf{ID}_{\mathcal{D}}$ is defined as follows:

- root node d_0 represents the root position ϵ, the topmost position in a term.
- if a node d represents a position such that there is a term t in the index with a $t_0(t_1, \ldots, t_n)$ at position d, then d has $n+1$ child nodes one for each $t_i, i = 0, \ldots, n$. The edges between d and its children are labelled by the relative positions $\mathsf{appl}_{n,j}$ for each t_j, $j \in \{0 \ldots n\}$, Here n represents the arity of the function.
- if d represents a position such that there is a term t in the index that contains a λ-abstraction $t' : \mathsf{abstr}$ at position d, it has a child for the position of the abstracted term. The relative position (i.e. the edge label) is abstr.

The node attributes are used to store information on the tree structure and recorded subterm occurrences. Again attributes named T indicate a link to the term graph, and D points to the positional tree.

- $d.D_{super}$ is the parent node of d
- $d.D_{sub}$ is a list of pairs (pos, d'), one for each child node d' and its relative position pos(i.e., abstr or $\mathsf{appl}_{i,j}$).
- $d.T_{this}$ is a list of all pairs (t, t_{sub}) such that there is some indexed term t with t_{sub} as subterm at position d.

Whenever a new term node in the term graph is created for a term t, the positional tree is updated (note that nodes are also created for subterms; e.g., during the insertion of $f((a+b)+c)$, both t_4 and t_5 are created and the positional tree is updated for both). This update records a pair (t, t_{sub}) in each position node d, where t_{sub} occurs at position d in t. The update algorithm starts by recording (t, t) in root position d_0 (as t is the subterm of itself in the topmost position), and proceeds by recursively recording pairs (t, t_{sub}) in the appropriate position nodes d:

(i) Insert (t, t_{sub}) in $d.T_{this}$. Update term node t by inserting d in $t.D_{this}$.
(ii) If t_{sub} is of the form $\lambda.t'$, then check if $(\mathsf{abstr}, d') \in d.D_{sub}$ for some child node d'. If it is not, create child node d', insert (abstr, d') in $d.D_{sub}$, and set $d'.D_{super} = d$. Proceed by recording (t, t') in d'.
(iii) If t_{sub} is of the form $t_0(t_1, \ldots, t_n)$, then check if $(\mathsf{appl}_{n,i}, d_i) \in d.D_{sub}$ for some child node d_i for each $i = 0, \ldots, n$. If it is not, create the child node d_i, insert $(\mathsf{appl}_{n,i}, d_i)$ in $d.D_{sub}$, and set $d_i.D_{super} = d$. Proceed by

recording (t, t_i) in d_i.

The positional tree shown in Figure 2 is the result of the insertion of the terms $\lambda.f(x_0)$, $f(a+b)$, and $f((a+b)+c)$ into an empty index. The content of the position node records $d.T_{this}$ is then:

$d_0.T_{this} = \{(t_0,t_0),(t_1,t_1),(t_2,t_2),(t_3,t_3),(t_4,t_4),(t_5,t_5)\}$

$d_1.T_{this} = \{(t_0,t_1)\}$

$d_2.T_{this} = \{(t_1,f),(t_2,f),(t_4,f)\}$

$d_3.T_{this} = \{(t_1,x_0),(t_2,t_3),(t_4,t_5)\}$

$d_4.T_{this} = \{(t_3,+),(t_5,+)\}$

$d_5.T_{this} = \{(t_3,a),(t_5,t_3))\}$

$d_6.T_{this} = \{(t_3,b),(t_5,c)\}$

$d_7.T_{this} = \{(t_0,f)\}$

$d_8.T_{this} = \{(t_0,x_0)\}$

$d_9.T_{this} = \{(t_2,+),(t_4,+)\}$

$d_{10}.T_{this} = \{(t_2,a),(t_4,t_3)\}$

$d_{11}.T_{this} = \{(t_2,b),(t_4,c)\}$

$d_{12}.T_{this} = \{(t_5,+)\}$

$d_{13}.T_{this} = \{(t_5,a)\}$

$d_{14}.T_{this} = \{(t_5,b)\}$

$d_{15}.T_{this} = \{(t_4,+)\}$

$d_{16}.T_{this} = \{(t_4,a)\}$

$d_{17}.T_{this} = \{(t_4,b)\}$

The position of occurrences are furthermore recorded in the nodes of the term graph:

$t_0.D_{this} = \{d_0\}$

$t_1.D_{this} = \{d_0,d_1\}$

$t_2.D_{this} = \{d_0\}$

$t_3.D_{this} = \{d_0,d_3,d_5,d_{10}\}$

$t_4.D_{this} = \{d_0\}$

$t_5.D_{this} = \{d_0,d_3\}$

$x_0.D_{this} = \{d_3,d_8\}$

$f.D_{this} = \{d_2,d_7\}$

$+.D_{this} = \{d_4,d_9,d_{12},d_{15}\}$

$a.D_{this} = \{d_5,d_{10},d_{13},d_{16}\}$

$b.D_{this} = \{d_6,d_{11},d_{14},d_{17}\}$

$c.D_{this} = \{d_6,d_{11}\}$

As an example how to interpret the above data consider the occurrence of term c. It appears in two positions d_6 and d_{11} given in $c.D_{this}$. In the former it is the second argument of a binary application, which comes from the insertion of the node $t_5 = ((a+b)+c)$ into the term graph. In the latter position where it is nested one positional level deeper, namely as subterm of $t_4 = f((a+b)+c)$. Both t_4 and t_5 can be found in the T_{this} attribute of the respective position. In the following section we describe how these records are used for fast retrieval of terms from the index.

2.3 Term Retrieval

To retrieve terms from the index, the information given both in the term graph and the positional tree is exploited in a three step algorithm. Given a query term t_q that possibly contains query variables, we

(i) retrieve an initial set of candidate terms,

(ii) remove all terms from the set for which no consistent query variable instantiation exists,

(iii) resolve all occurrences of the special variable Φ in t_q, that is, if $\Phi(t'_q)$ occurs at a position d in t_q, then check for each candidate term that they have a subterm at d that contains an occurrence of t'_q.

Observe that in step iii the term t'_q can again be a query term containing query variable and instances of Φ, thus its retrieval spawns are recursive calls to the retrieval algorithm.

Step i

We identify subterms of t_q that contain no query variables and their position in t_q. For each subterm t' occurring at position d, indexed terms with corresponding subterms can be looked up in $d.T_{\mathit{this}}$. For each entry (t_c, t') in $d.T_{\mathit{this}}$, t_c is a candidate term for the query. By intersection of the sets of candidate terms obtained for each of those subterms t', the initial set of candidates is obtained.

Consider the example $t_q = f(\alpha + \beta)$, where $\alpha, \beta \in \mathcal{Q}$ are query variables. The only subterms not containing any query variables are f and $+$ at position d_2 and d_9, respectively (cf. Figure 1). For f, the terms $\{t_1, t_2, t_4\}$ are retrieved from $d_2.T_{\mathit{this}}$, and for $+$ the terms $\{t_2, t_4\}$ are retrieved from $d_9.T_{\mathit{this}}$. The intersection of both sets $\{t_1, t_2, t_4\} \cap \{t_2, t_4\} = \{t_2, t_4\}$ yields the initial set of candidate terms for the query, whose members are $t_2 = f(a + b)$ and $t_4 = f((a + b) + c)$.

Step ii

We then compute the query variable instantiations and check their consistency for each term t_c in the candidate set. An instantiation for a query variable is given by the subterm in the corresponding position in t_c. If a query variable has multiple occurrences in t_q, it must have the same instantiation for all its instantiations in t_c. If not t_c is removed from the candidate set.

For the candidates from the previous step, the variable instantiations are $\{a/\alpha, b/\beta\}$ for $t_2 = f(a+b)$ and $\{(a+b)/\alpha, c/\beta\}$ for $t_4 = f((a+b)+c)$. Both α and β occur only once, so there is no conflict to resolve. However, if we have $f(\alpha + \alpha)$ as query term, Step i would still compute the same candidate set, but no consistent variable instantiation could be derived from the candidates and the query would fail.

Step iii

For each occurrence of the special query variable $\Phi(t'_q)$ in t_q and each candidate term t_c we have obtained a valid instantiation $t'_c/\Phi(t'_q)$ in step ii. Let $t'_1, \ldots, t'_n$ be the terms retrieved from the index on the query t'_q. Then some $t'_i \in \{t'_1, \ldots, t'_n\}$ must be a subterm of t'_q, if t_c matches the query. In this case there is a $d \in t'_i.D_{\mathit{this}}$ where $(t'_c, t'_i) \in d.T_{\mathit{this}}$. Otherwise t_c does not match the query.

Take $t_q = f(\Phi(b))$ as an example query. In this case, the candidate set after executing Step i and Step ii is $\{t_1, t_2, t_4\}$ and the instantiations for $\Phi(b)$ are $\{x_0, t_3, t_5\}$. The query $t'_q = b$ retrieves b. Next the set of terms with occurrences of b are looked up in $d_i.T_{\mathit{this}}$ for $d_i \in b.D_{\mathit{this}} = \{d_6, d_{11}, d_{14}, d_{17}\}$, the result is $\{t_3, t_2, t_5, t_4\}$. The condition described above is thus fulfilled for $t'_c \in \{x_0, t_3, t_5\} \cap \{t_3, t_2, t_5, t_4\} = \{t_3, t_5\}$, so the terms retrieved for query $f(\Phi(b))$ are the super-terms of t_3 and t_5 which are $t_2 = f(a+b)$ and $t_4 = f((a+b)+c)$, respectively.

All terms t_c remaining in the candidate set after Step iii has been executed match the query t_q and are returned. The algorithm implements *forward* *matching*, i.e. the retrieval of all indexed terms matching a query t_q. In the next section we describe a mechanism for *backward matching*, where queries are stored in the index and matching queries are identified at the insertion of new terms.

3 Meta Nodes and Reactive Behaviour

Generally, in a single proof attempt, a fixed number of CAS algorithms is employed. Thus we have a static set of query terms that form the abstract description of these algorithms. On the other hand, the set of terms in the proof changes during proof development. This suggests to store queries permanently and to find matching queries at the time a new indexed term is added. The index operates now as a term filter, and matching queries are returned, indicating that the algorithm related to a query may be applicable now.

The mechanism we describe is a 'lazy' version of the retrieval algorithm in section 2.3. Query terms are stored in an additional graph structure $\mathsf{ID}_{\mathcal{M}}$ of reactive meta nodes. Their task is to monitor updates on the positional tree and wait for the insertion of specified term fragments in some monitored positions. Once a suitable term has been inserted the meta node initiates an action on that term such as a call to a simplification algorithm. This scenario is achieved by adapting in particular Steps i and iii of our retrieval algorithm from Section 2.3.

Every meta node $m \in \mathsf{ID}_{\mathcal{M}}$ encodes a query term t_q. Analogously to Section 2.3, all subterms t'_q that contain no query variables are determined. For each of these subterms t'_q, a connection between m and the positional node d corresponding to the term position of t'_q in t_q is established with the following node attributes:

- $m.D$ contains all positions m is associated to, for each subterm t'_q.
- $d.M$ is an additional attribute for nodes in the positional tree, containing a list of pairs (t'_q, m) indicating that meta node m waits for the insertion of term t'_q.
- $m.V$ contains all query variables in t_q and the positions at which they occur. Thus entries are pairs $(\chi, \{d_1 \dots d_n\})$ for every query variable χ that occurs in positions $d_1 \dots d_n$.
- $m.A$ is an action that is carried out once a suitable term matching t_q has been entered.

The procedure to update the positional tree sends a *notification* $\texttt{notify}(d, t'_q)$ to a meta node m whenever a term t'_q is recorded in position d and $(t'_q, m) \in d.M$. We say m waits for t'_q at position d. If some meta node m has received notifications from all position nodes $d \in m.D$ after a new term t has been added to the index — this is equivalent to t being identified as a candidate term in Step i of the retrieval algorithm — we say meta node m has been *triggered*. Note that only notifications caused by the same term t are relevant, thus the record of sent notifications is cleared after each update of the positional tree. Once m is triggered, the instantiations for the query variables in $m.V$ are computed and their consistency is checked, analogously to Step ii of the retrieval algorithm.

Consider again our example query $f(\alpha+\beta)$. Its associated meta node m has the attributes $m.D = \{d_2, d_9\}$ and $m.V = \{(\alpha, \{d_{10}\}), (\beta, \{d_{11}\})\}$. Furthermore, the pair (f, m) is inserted in $d_2.M$, and $(+, m)$ in $d_9.M$. If now the term $f(a+b)$ is inserted into the index, f is recorded in d_2, and $+$ in d_9. Both nodes send a notification to m, as no further positions are recorded in $m.D$, m is triggered and the consistency of the variable instantiations is successfully checked.

To resolve occurrences of the special query variable Φ, if there are any, Step iii of the retrieval algorithm in Section 2.3 has to be modified. For each $\Phi(t'_q)$ in t_q we can predetermine terms t' matching t'_q and store them in an additional meta node m'. Once the meta node m is triggered by a newly inserted term t matching t_q except for the resolution of occurrences of Φ, we check whether t' occurs at a suitable position in t. To describe dependencies between these meta nodes and to collect predetermined instantiations we define three further node attributes:

- $m.I$ collects terms matching the query term. Entries are $(t, \{(\chi_1, t_1)\dots(\chi_i, t_i)\})$ for every term t matching t'_q where every query variable χ_j in t'_q is instantiated with t_j. When new terms are entered into the index m can find new suitable terms and enter them into $m.I$.
- $m.M_{super}$ is the list of meta nodes m_{super} that are notified whenever a new term is found by m and inserted in $m.I$.
- $m.M_{sub}$ is the list of all meta nodes m depends on. Entries are (m_{sub}, d) for every meta node m_{sub} modelling a query for subterms that occur at an arbitrary position below d.

The attributes $m.M_{sub}$ and $m.M_{super}$ form a trigger mechanism similar to that formed by $m.D$ and $d.M$, the notion of a meta node m being triggered is extended to m having received a notification from all position nodes $d{\in}m.D$ and from all meta nodes $m'{\in}m.M_{sub}$ it depends on. A meta node m sends a notification to all meta nodes in $m.M_{super}$, whenever it has found a new instantiation $(t, \{(\chi_1, t_1)\dots(\chi_i, t_i)\})$ which is recorded in $m.I$.

At the time m is triggered by a term t, some of the $m'{\in}m.M_{sub}$ may have found several instantiations $(t', \{(\chi_1, t_1), ..., (\chi_n, t_n)\}){\in}m'.I$. Suitable instantiations among these are determined by:

(i) Checking that the query variable instantiations are consistent. That is, for each instantiated variable χ_i that is also instantiated by m both instantiations are consistent. Otherwise the instantiation is dropped.

(ii) For each of the remaining instantiations $(t', \{(\chi_1, t_1), ..., (\chi_n, t_n)\})$, it is evaluated if there is a term t_c such that $(t_c, t'){\in}d.T_{This}$ for some $d{\in}t.D_{This}$, where $(t, t_c){\in}t_d.D_{This}$, i.e. if t' is a subterm of t below the specified position d. This is done by two intersections of term sets analogous the one presented in Section 2.3, respectively one intersection and one membership test, as one of the sets is the singleton $\{t_d\}$.

Once a term has been inserted and an meta node was triggered successfully, the action of that meta node is carried out on the term.

As an example, we will describe a net of two meta nodes encoding the query $f(\varphi(\Phi(\varphi(a,\beta)),\gamma))$, i.e. a term $f(\varphi(\Phi,\gamma))$ where somewhere in Φ there is an occurrence of the subterm $\varphi(a,\beta)$, and the function symbol φ is the same in both occurrences. The query is encoded as follows:

A first meta node m_1 is employed to wait for the whole term. Since f is the only symbol that is not a query variable and it is found in position d_2 we get $m_1.D = \{d_2\}, d_2.M = \{(f, m_1)\}$. The query variables φ, denoting a binary function, and γ can be instantiated by the subterms in positions d_9 and d_{11}, which is stored as $m_1.V = \{(\varphi, \{d_9\}), (\gamma, \{d_{11}\})\}$. Finally the special query variable Φ indicates a subterm that has to occur below position d_{10} which has to match $\varphi(\alpha,\beta)$. This is captured in a second meta node m_2 which is connected to m_1 via $m_1.M_{sub} = \{(m_2, d_{10})\}$ and $m_2.M_{super} = \{m_1\}$. Symbols and variables are defined analogously to m_1, here a has to occur in position d_5, φ in d_4 and β in d_6, thus we get $m_2.D = \{d_5\}, d_5.M = \{(a, m_2)\}, m_2.V = \{(\varphi, \{d_4\}), (\beta, \{d_6\})\}$.

Suppose now the term $t_4 = f((a+b)+c)$ is inserted into the indexing structure. Then first node m_2 will be be triggered by t_3, as a is inserted at position d_5 with root term t_3. Therefore, d_5 sends a notification $\texttt{notify}(d_5, t_3)$ to m_2. As this is the only position m_2 is waiting for, the variables are instantiated according to $m_2.V$ and we get $m_2.I = \{(t_3, \{(\varphi, +), (\beta, b)\})\}$. Furthermore, m_1 is notified that m_2 has found a matching term. Next m_1 will be triggered when the new node t_4 is created. Here d_2 will send the notification $\texttt{notify}(d_2, t_5)$. As d_2 is the only position m_1 is waiting for variables are instantiated here, too: $m_1.I = \{(t_4, \{(\varphi, +), (\gamma, c)\})\}$. Since the only sub-node m_2 has already sent a notification, we check that the variable instantiation of m_2 is suitable.

First the query variable instantiations are checked, φ is the only variable that is instantiated in both nodes, and it is in both cases bound to $+$. Thus it is evaluated whether t_3 is at a suitable term position in t_5. The term t_3 occurs in positions d_0, d_3, d_5 and d_{10}, the root terms which are recorded with subterm t_3 in these positions are t_3, t_2, t_5 and t_4, respectively. As m_2 is associated with d_{10} in m_1, the subterm of t_4 in d_{10} has to be looked up. It is t_3, and as $t_3 \in \{t_3, t_2, t_5, t_4\}$, the term identified by m_2 is a suitable subterm in a suitable position. Thus m_1 has identified a term matching pattern $f(\varphi(\Phi(\varphi(a,\beta)),\gamma))$ along with the suitable query variable instantiations: $(t_4, \{(\varphi, +), (\beta, b), (\gamma, c)\})$

In case a term does not match, as for example with $t_2 = f(a+b)$ in figure 1, the matching will fail at some point. For t_2, both nodes m_1 and m_2 will be triggered: m_1 by t_2 and m_2 by $t_3 = (a+b)$. The variable instantiations are consistent, as again φ will be bound to symbol $+$. The last step, however, to determine if t_3 is at a suitable term position, will fail, because there is no entry (a, t_3) at any position $d \in t_3.D_{This}$ with root term a and subterm t_3. This is the worst case of a failing matching, in general the unsuccessful branches of the algorithm should be cut at earlier stages.

4 Aim and Application

The mechanism presented here is intended to be applied at the frontier between deduction system and CAS. While the integration of an external CAS's result with the CAS operating as a blackbox have been explored in several experiments [1,3], the verification of these CAS results within the deduction system's formalism is still difficult. Especially the operation of CASs on efficient data structures, which support e.g. an optimized treatment of commutative and associative operators, makes the syntactical transformation of terms hard to track.

A solution is a verification through a *traceable* CAS [13,11]. The CAS is here employed for term simplification in first place. The output of the CAS, the input being the term to simplify, is the simplified term and the *trace*, which is a sequence of rewrite steps to transform the original term into its simplified form. Based on this trace, the transformation of a term into its simplified form can be reconstructed and thus verified by a deduction system. As the focus of the CAS is here on explicitly traceable computations, the CAS has to operate on data structures which are close to the syntax of the interacting deduction system. Especially the treatment of commutativity and associativity has to be explicitly traceable (in the experiments described in [13] and [11], most of the trace was concerned with commutative and associative rewriting). Thus the functionality of the a CAS is focused on *structural* rewriting.

The central control structure of the CAS is a blackboard architecture implemented by the indexing method presented here. Fresh terms to be processed by the CAS are either provided by the deduction system it interacts with or are results of the application of algorithms. Every time a fresh term is inserted into the blackboard, algorithms that may have become applicable are detected. Algorithms and their applicability are described in *trigger rules*. These trigger rules are characterized by a query term and an action. Query terms are represented by meta nodes, matching query terms are identified as described in section 3. Simplified examples for trigger rules are (in the form *query* $\longrightarrow$ *action*):

(i) $\Phi(\texttt{normalise}(\alpha+\beta)) \longrightarrow \Phi(\texttt{normaladd}(\texttt{normalise}(\alpha), \texttt{normalise}(\beta)))$
(ii) $\Phi(\texttt{normalise}(\alpha \cdot \beta)) \longrightarrow \Phi(\texttt{normalmult}(\texttt{normalise}(\alpha), \texttt{normalise}(\beta)))$
(iii) $\alpha+\beta = \Phi(\alpha) \longrightarrow \alpha+\beta = \texttt{pop}(\Phi, \alpha, \mathit{pos})$
(iv) $\alpha^3+\beta\cdot\alpha^2+\gamma\cdot\alpha+\delta = 0 \longrightarrow \texttt{solvecubic}(\beta, \gamma, \delta)$

First thing to be noticed is that *keywords* such *mathttnormalise* or `normaladd`, which are not part of the syntax of the deduction system's syntax, can be integrated into the term to guide the processing of terms. By simply inserting the term describing the action, simple systems of rewrite rules can be easily implemented. Assuming that `normaladd`(α, β) and `normalmult`(α, β) will be rewritten by the result of the addition respectively the multiplication of polynomials in normal forms, the rules 1 and 2 implement the structural recursion of normalisation of polynomials.

The action performed by rule 3 is slightly more complex. It is used to establish syntactic equality of two terms by commutative and associative permutation. Here the term α is moved to the top of term $\Phi(\alpha)$, the keyword for this is `pop`. In this case, the new term is nopt simply added to the blackboard, but an algorithmic procedure to implement `pop` is provided suitable arguments Φ,α and *pos*, the position of α in Φ, and performs the necessary commutative and permutative rewrite steps to move α to the head position of Φ.

Rule for is an example of the integration of a (possibly external) specialized algorithm into the mechanism. In this case an algorithm to resolve cubic equations is called, its arguments are the coefficients of the polynomial $\alpha^3 + \beta\cdot\alpha^2+\gamma\cdot\alpha+\delta$. The result term to be inserted, if any, is entirely produced by the external algorithm. This is an example of an application of the blackboard as a broker architecture, e.g. to coordinate a collection of specialised algorithms or even a number of external systems.

The key features of the indexing method presented in this paper are clearly motivated by the structure of these trigger rules. Its syntax is simple and a type system is omitted, as the matching to be performed is purely structural. This allows for pragmatic special features like the use of keywords which are outside the formalism of the connected deduction system, but are useful in a simple and intuitive implementation of trigger rules. An important features is the use of special variables Φ as in rule 3, this is a useful tool when dealing with commutative and associative permutation of terms. The basis of the blackboard's functionality is the net of reactive meta nodes described in section 3 which allows to evaluate exhaustively a number of trigger rules with reasonable computational overhead.

Unfortunately, a thorough evaluation of the method's efficiency when processing large examples has not been undertaken yet. However, experiments in related work indicate a reasonable performance. The indexing method presented here is a variant of coordinate indexing and its refinement path indexing. The performance of these methods has been evaluated by McCune [4] and compared to discrimination tree indexing. While discrimination tree indexing outperformed path indexing in his experiments, path indexing respectively coordinate indexing as a basis is nevertheless a reasonable choice for our purposes. Path indexing can be expected to perform well for instance retrieval, which is the main operation here. The retrieval of generalisations, where discrimination tree indexing was significantly faster in McCune's experiments, is tackled in a different way in our approach (see section 3): Query terms are stored in a separate graph structure and matching is evaluated in a lazy fashion. The intersection of term sets is replaced by the collection of notifications, which can be efficiently implemented by bit vectors. A triggered meta node can thus be detected with minimal computational overhead. Furthermore, path indexing can be expected to perform well, if there are many variables in the query term, respectively if there are few constant symbols. The query terms of trigger rules have usually a moderate number of constant symbol occurrences here. In first place however, path indexing can be adapted to deal with commutativity and associativity, assuming that the effect is a permutation of subterm positions in equal terms. In our approach, this idea is elaborated in the retrieval of query terms featuring special variabels with occurrences of subterms at unspecified positions.

5 Conclusion

We have presented an indexing method which is an extension of coordinate indexing as described by Stickel [12]. We have added a graph for shared representation of terms, which allowed us to implement operations on sets of terms as operations on integer sets and reduces redundancies in term retrieval. As a further improvement we have added a reactive term filtering mechanism that enables for every new term that is added to a proof state to immediately detect if particular algorithms, such as simplifications, are applicable. This functionality is especially useful when interfacing to CAS algorithms. A set of application criteria for the integrated algorithms can be specified as query terms and pre-compiled in a net of reactive meta nodes, guaranteeing their automatic application whenever possible. Moreover, as opposed to other indexing techniques in the literature, our approach allows for inside out search to retrieve terms that have occurrences of instances of query terms in arbitrary positions, which is useful when a CAS is applied inside a rewrite step to simplify or normalise subterms in a formula.

While we have demonstrated the indexing technique only with a small abstract example in this paper, the employed higher order language including λ-abstraction provides a versatile means to express the input syntax of a diverse range of algorithms. The indexing technique was developed for the reimplementation of ΩMEGA's whitebox integration of a CAS that provides a collection of arithmetic simplification algorithms [13] and it is used there successfully as an automatic algebraic rewrite system. However, an integration of the technique into a more advanced ATP system such as ΩMEGA's proof planner is intended.

While the indexing seems to improve performance, an evaluation of the computational complexity of an indexing technique is in general difficult, especially asymptotic worst-case or average-case complexity analysis is not useful in practise [5]. For the higher order case there are few comparable approaches, which furthermore differ strongly in the conditions for term retrieval, e.g. perfect versus imperfect filtering or simplification of higher order aspects. Unlike the approach by Pientka [7] the aspects of higher order unification are neglected and a first order style matching is employed instead, however higher order terms containing λ-abstractions can be indexed.

Ednote: das ist neu: The indexing method presented here is work in progress. In future work, especially the actual performance of the method when processing large example problems and possible optimization of implementational details have to be evaluated.

References

[1] C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In *Proc. of ISSAC'95*, p. 150–157. ACM Press, 1995.

[2] 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.

[3] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

[4] W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[5] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. On the evaluation of indexing techniques for theorem proving. In *Proc. of IJCAR 2001*, LNAI 2083, p. 257–271. Springer, 2001.

[6] L. Paulson. Isabelle reference manual. Tech. Report, University of Cambridge, 2005.

[7] B. Pientka. Higher-order substitution tree indexing. In *Proc. of ICLP 2003*, LNCS 2916, pages 377–391. Springer, 2003.

[8] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *Journal of AI Communications*, 15(2-3):91–110, 2002.

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

[10] J. Siekmann, *et al.* Proof development with Ωmega. In *Proc. of CADE-18*, LNAI 2392, p. 144–149. Springer, 2002.

[11] V. Sorge. Non-trivial symbolic computations in proof planning. In *Proc. of FroCoS 2000*, LNCS 1794, p. 121–135. Springer, 2000.

[12] M. Stickel. The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, 1989.

[13] F. Theiß. On the White Box Integration of Computer Algebra Algorithms into a Deduction System. Master's thesis, Universität des Saarlandes, November 2005.

[14] Ch. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, Ch. Theobald, and D. Topic. SPASS version 2.0. In *Proc. of CADE-18*, LNAI 2392, p. 275–279. Springer, 2002.