Chapter 26

Term Indexing

I.V. Ramakrishnan

R. Sekar

Andrei Voronkov

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

1. Introduction

1.1. Motivation

First-order terms constitute the basic representational unit of information in several disciplines of computer science such as automated deduction, term rewriting, symbolic computing, and logic and functional programming. Computation is done by operations germane to manipulating terms such as unification, pattern matching, and subsumption. Often these operations are performed on collections of terms. For instance, in logic programming, deductive databases, and theorem-proving by model elimination we need to select all candidate clause-heads in the program that unify with a given goal. In automated deduction, term rewriting and functional programming, the selection criteria may be based on unifiability (e.g., in resolution), matching (e.g., in normalization) or subsumption. Such retrieval of *candidate terms* that bear a specific relationship to a given *query term* (or set of query terms) is a central operation in automated theorem provers, deductive databases, and logic and functional programming systems. In the absence of techniques for speeding up the retrieval of candidate terms, the time spent in identifying candidates may overshadow the time spent in performing other useful computation. The problem is especially important in contexts where the data sets get large and/or keep growing, as in automated theorem provers and deductive databases. Clearly, a naive approach based on linear search through the set of terms degrades very quickly when large data sets are involved. It is fast enough to start with, but as the term set grows, more and more time is spent in the search for suitable candidates. This leads to a rapid and monotonic drop in performance of the system, as pointed out by Wos [1992]:

> *"After a few CPU minutes of use, a reasoning program typically makes deductions at less than 1% of its ability at the beginning of a run."*

This factor has led to a lot of research interest in *term indexing* techniques, which refers broadly to techniques for the design and implementation of structures that facilitate rapid retrieval of a set of candidate terms satisfying some property (such as generalizations, instances, unifiability, etc.) from a large collection of terms. Use of term indexing techniques have resulted in dramatic speed improvements, ranging from one to several orders of magnitude, in all major theorem provers, including (in the alphabetical order) Bliksem [de Nivelle 2000], E [Schulz 1999], Fiesta [Nieuwenhuis, Rivero and Vallejo 1997], Gandalf [Tammet 1997], Otter [McCune 1994*a*, McCune and Wos 1997], Setheo [Moser, Ibens, Letz, Steinbach, Goller, Schumann and Mayr 1997], Snark [Stickel, Waldinger, Lowry, Pressburger and Underwood 1994, Stickel, Waldinger and Chaudhry 2000], Spass [Weidenbach, Gaede and Rock 1996], Vampire [Riazanov and Voronkov 1999], and Waldmeister [Hillenbrand, Buch, Vogt and Löchner 1997]. Term indexing techniques enable theorem provers to continue performing deductions at a steady pace, as opposed to the rapid degradation observed in the absence of term indexing. For instance, Wos [1992] observed that "... Otter makes deductions at the rate of 550 per second in the first few seconds, and at the rate of 460 per second after 19 CPU hours ...".

Hiper runs 20 to 30 times faster on small and moderate sized problems, with the asymptotic speedup approaching infinity [Christian 1993]. The benefits are less dramatic in functional and logic programming, but nevertheless very significant — use of indexing leads to typical improvements in speeds of such programs by between 20% to 200%. Effective term indexing techniques have hence become an integral component of high-performance declarative programming and automated reasoning systems.

Since the early results demonstrating the effectiveness of indexing techniques, research into term indexing has acquired major momentum. In particular a variety of new techniques were invented and implemented for term indexing. One reason for developing so many techniques is that the conditions under which terms are retrieved differ for different operations. For instance in pattern matching we have to retrieve terms that are generalizations of an input term, whereas for generating critical pairs we retrieve terms that unify with the input term. In addition, indexing algorithms tend to exhibit tradeoffs in retrieval speed and space usage. Thus no one indexing technique can cater to all applications.

This paper presents a survey of the main indexing techniques that have been developed in the past. We formulate these techniques within a uniform framework that makes it easier to understand the advantages, disadvantages and trade-offs in developing and using term indexing.

1.2. Formulation of the term indexing problem

The problem of term indexing can be formulated abstractly as follows. Given a set $\mathcal{L}$ (called the *set of indexed terms*), a binary relation R over terms (called the *retrieval condition*) and a term t (called the *query term*), identify the subset $\mathcal{M}$ of $\mathcal{L}$ consisting of all of the terms l such that $R(l,t)$ holds. If $R(l,t)$ holds, we say that l is *R-compatible* with t, or simply *compatible* when R is clear from the context.

In some applications it is enough to search for a superset of $\mathcal{M}$, i.e., to also retrieve terms l for which $R(l,t)$ does not hold, but we would naturally like to minimize the number of such terms in order to increase the effectiveness of indexing. In other applications, it is acceptable to retrieve only some R-compatible indexed terms, i.e., a subset of $\mathcal{M}$. If the set of the retrieved terms is guaranteed to coincide with the set of R-compatible terms, then the indexing technique is said to perform *perfect filtering*. Otherwise, indexing performs *imperfect filtering*.

In the context of term indexing, it is usually the case that the relation R of interest is such that $R(s,t)$ holds if there exists substitutions σ and β such that $s\sigma = t\beta$, and furthermore, these substitutions satisfy certain additional constraints. For instance, if σ and β are constrained to be renaming substitutions, the relation $R(s,t)$ simply becomes a variant check. Likewise, if β is constrained to be the empty substitution, the relation $R(s,t)$ becomes an instance check. In addition to identifying R-compatible terms, we sometimes need to compute the substitutions σ and β as well.

The principal parameters associated with term indexing are:

- *Retrieval condition*, expressed by the relation R that determines the subset $\mathcal{M}$ of the indexed terms that need to be identified. The most common examples of retrieval condition are unification, matching, subsumption, etc.
- *Retrieval mode*, which determines whether the entire set $\mathcal{M}$ is returned, or whether the elements of this set are returned one at a time. In some cases we are only interested in nonemptiness of the candidate set.

1.3. Early research in indexing

Traditional notions of indexing. In very general terms, *indexing* refers to the ability to quickly filter out a set of candidate elements that satisfy specific criteria from a (typically) large data set. One of the oldest examples include card indexes in libraries, where the data set consists of all the books in the library, and the selection criteria may be based on author names, titles or keywords. Another example is that of indexes in books, where the data set includes all the pages in the book, and the selection criteria is based on the appearance of keywords in a page. A more formal treatment of indexing was developed in the context of databases, where the data set consisted of all the records in the database, and the selection criteria were based on the values of one or more of the fields in the record.

This paper deals with the same general problem as captured by the above examples, but in the context of automated reasoning, declarative programming and deductive databases. In these contexts, the principal data of interest are *first-order terms*, which are much more expressive and complex as compared to the simple data values that arise in text indexing or databases. Whereas selection criteria are traditionally based on single attributes (i.e., retrieval of all records that have the specified value of an attribute), for terms, it is based on complex operations such as unification and matching on these terms. In addition, these operations may have to be performed in the context of an equational theory, the most common such theory arising in the context of AC symbols. Finally, we may be interested in multiterm indexing problems discussed later.

Attribute-based indexing. In *attribute-based indexing*indexing!attribute-based, we map some features of a term t into a simple-valued (say, integer-valued) attribute a_t. Indexing is then based on identifying a relation R_A on attributes of t and s such that $R(s,t) \Rightarrow R_A(a_s, a_t)$ (or $R_A(a_s, a_t) \Rightarrow R(s,t)$). For instance, if the retrieval relation is *inst*, then the attribute can be the number of function symbols in a term. (Observe that if t is an instance of s then the number of function symbols in t is greater than or equal to that of s.) Consider several examples of attribute-based indexing.

- *Matching pretest* makes use of the fact that for a term t to be an instance of l, the number of symbols in t is greater than or equal to the number of symbols in l.
- *Outline indexing* makes use of the fact that t and l are unifiable only if they

agree at all nonvariable positions. Employs a bit-vector to encode the nonvariable positions and the corresponding symbols [Henschen and Naqvi 1981].

- *Superimposed codewords*: the attribute is obtained by logical-or operations on the bit representations of function symbols at specified positions within a term [Wise and Powers 1984, Ramamohanarao and Shepherd 1986]

Attribute-based indexing is based on the assumption that a relation involving simple-valued attributes is much easier to compute than performing term matching or unification. Thus, it can be used to coarse filter for the likely candidates. However, it has several disadvantages. Firstly, the accuracy of attribute-based indexing is typically low. Second, if the index set is large, the coarse filter, while an improvement over the naive approach of matching or unifying with every term in the indexed set, may still be inefficient as it may involve checking the relation R_A for each term in the set.[1] Due to these disadvantages, we will focus on symbol-based indexing for the rest of this paper.

Function symbol based indexing. The retrieval condition is typically based on identification of a unifying substitution between the query and indexed terms, with various constraints placed on the substitutions. Thus, the question of whether the retrieval condition holds between the query term and an indexed term is determined by the function symbols in both these terms. For instance, in every position where both the query term and candidate term contain a function symbol, these symbols must be identical. Therefore we can make use of some or all of the function symbols in the indexed terms in determining the candidate terms. Most known term indexing techniques are based on this observation, and we refer to such techniques broadly as *function symbol based indexing*, or simply as *symbol-based indexing*. The rest of this paper presents a survey of some of the most important symbol-based indexing techniques.

1.4. Overview of organization

This paper is organized into five parts.

1. In Section 2, we provide the requisite background, including notations and definitions. We formulate the term indexing problem and its context. In Section 3 we outline the basic representation and implementation techniques related to terms and indexing.
2. In Section 4 we show how term indexing can be formulated in terms of string matching. The common framework enables us to understand many indexing methods as instances of the generic framework. The framework also makes it

[1] It would be desirable to choose attributes and the relation on the attribute such that the candidate terms can be identified quickly, without having to try all of terms in the indexed set. For instance, in the case of the *inst* relation, we can store the index set as an array (or tree) that is sorted on the value of the size attribute. We can then perform a search in $O(\log n)$ time to identify all terms that are larger than the query term.

easier to compare and contrast known approaches. Most importantly, the framework distills out essential characteristics of different techniques as elaborated in Section 16.

3. The third part of this paper presents a survey of many indexing techniques: path indexing (Section 5), discrimination trees (Section 6), adaptive automata (Section 7), automata-driven indexing (Section 8), substitution trees (Section 10), unification factoring (Section 12), code trees (Section 9), and context trees (Section 11). To keep this part down to a reasonable size, not all known methods are surveyed. Instead, we have made an effort to capture diverse methods. Particular emphasis have been given to works that provide a formal treatment of space/time complexity and optimality.
4. The fourth part of this paper treats some advanced indexing techniques: multi-term indexing (Section 13), issues in perfect filtering (Section 14), and indexing modulo the AC-theory (Section 15).
5. The fifth (and the last) part of this paper summarizes the indexing techniques considered so far, discusses implementations of indexing in logic and functional programming, and also in theorem provers, and discusses new directions for term indexing. In Section 16, we summarize the indexing techniques considered so far by providing a list of "basic elements" of term indexing techniques. Many of these elements are in fact based on concepts well known in string-matching automata. The common framework developed in Section 4 enables us to lift these concepts from the domain of string matching to the domain of term-indexing. We describe each of these elements and the issues and trade-offs in employing them in indexing methods. We also provide a summary of how these elements have been combined in indexing methods proposed so far. In Section 17 we briefly describe indexing as implemented in some declarative programming systems and theorem provers. In Section 18 we sketch possible new directions for indexing.

2. Background

2.1. Notations and definitions

We begin this section with the notations and concepts used in the rest of this paper. We assume familiarity with the basic concept of a *term*. The symbols in a term are drawn from a nonempty *alphabet* Σ and a countable set of variables $\mathcal{V}$. A term is *ground* if it contains no occurrences of variables. With each symbol s in the alphabet is associated a nonnegative integer, called the *arity* of s, denoted $arity(s)$. We assume that the terms are well-formed, i.e., that the every symbol in the term has the correct number of arguments, as given by the arity of the symbol. We will use a, b, c, d and f to denote nonvariable symbols (which are sometimes referred to as *function symbols*) and x, y, z (with or without subscripts and primes) to denote variables. We also denote variables using a wildcard symbol $*$, with or without subscripts. The symbol $*$ written without subscripts will have two uses, depending on context. In some contexts it means an occurrence of a unique variable. For example, $f(x, x, *, *)$ denotes any term of the form $f(x, x, y, z)$, where y, z are distinct variables different from x. In other contexts it will be used as a wildcard symbol denoting any variable. $root(t)$ denotes the *top symbol* of t, i.e., the symbol appearing at the root of a term t. We will also (somehow ambiguously) use the wildcard ? in terms. This wildcard will always denote a term whose top symbol is different from a certain set of function symbols, depending on the context.

In order to refer to subterms of a term, we develop the following concept of a *position*[2]:

2.1. DEFINITION *(Position).* A *position* is either the empty string Λ, or $p.i$, where p is a position and i an integer. The notions of a *position in a term* and the *subterm of t at a position p*, denoted t/p, are defined as follows.

- Λ is a position in t and $t/\Lambda = t$;
- If $t/p = f(t_1, \dots, t_n)$, where $n > 0$, then $p.1, \dots, p.n$ are positions in t and $t/p.i = t_i$, for all $i \in \{1, \dots, n\}$.

Instead of $\Lambda.i$ we will write simply i. We use $\mathcal{P}$ to denote the set of all positions. The notation $\mathcal{P}(t)$ denotes all the positions in a term t, and $\mathcal{P}_v(t)$ and $\mathcal{P}_f(t)$ denote the subset of these positions at which t has variables and function symbols respectively. The set $\mathcal{P}_v(t)$ is called the *fringe* of t. We use $t[s]_p$ to denote the term obtained from t by replacing t/p by s.

We illustrate these concepts using the term $t = f(a(x), b(a(y), c))$. Here, $t/\Lambda = t$, $t/2 = b(a(y), c)$, $t/2.1 = a(y)$, and $t/2.2 = c$. The term $t[c]_2 = f(a(x), c)$ is obtained by replacing the second argument of f by (the term) c. The fringe of t is $\{1.1, 2.1.1\}$.

A *substitution* is a mapping from variables to terms. Given a substitution β, we denote by $t\beta$ the term obtained by replacing every variable x in t by $\beta(x)$. We say that t is an *instance* of s if $s\beta = t$ for some substitution β. If t is an instance of u then we write $u \leq t$ and call u a *prefix* of t. The inverse of $\leq$ relation is denoted by $\geq$.

We denote by $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ the substitution β defined as follows:

$$\beta(x) = \begin{cases} t_i, & \text{if } x = x_i; \\ x, & \text{otherwise.} \end{cases}$$

For example, for the term $t = f(a(x), b(y, z))$ and the substitution $\beta = \{x \mapsto b(x', x''), y \mapsto c\}$ we have $t\beta = f(a(b(x', x'')), b(c, z))$.

Terms t and s are called *unifiable*, if there exists a substitution β such that $s\beta = t\beta$.

[2]The terminology *occurrence* and *path* are sometimes used in the literature to denote the same concept.

2.2. The problem context

In this section, we describe the term indexing problem in the context of theorem proving, logic programming and deductive databases, and functional programming.

2.2.1. Theorem proving

First-order theorem provers can generally be divided in two kinds. The first kind is the *saturation-based* provers that implement various kinds of resolution or the inverse method. The second kind can be characterized as the *tableau-based* provers that implement semantic tableaux or model elimination.

Saturation-based provers. Examples are Fiesta [Nieuwenhuis et al. 1997], Gandalf [Tammet 1997], Otter [McCune 1994*b*], Spass [Weidenbach et al. 1996], Vampire [Riazanov and Voronkov 1999], and Waldmeister [Hillenbrand et al. 1997]. Such provers generate new clauses from a given set of clauses using suitable inference rules (binary resolution, hyperresolution, etc.) and add the new clauses to the set of already inferred ones (i.e., *saturate* a set of clauses under applications of the inference rules). More information on saturation-based provers can be found in [Lusk 1992, McCune 1994*b*, Riazanov and Voronkov 2000*a*] and [Weidenbach 2001] (Chapter 27 of this Handbook).

In addition to inference rules, saturated-based provers also use *simplification rules* which either "simplify" clauses in the search space (i.e., replace them by "simpler" clauses) or remove them from the search space completely. There is a variety of retrieval conditions used to identify applicability of inference or simplification rules, including instance, generalization, unifiability, miltiliteral forward and backward subsumption, and variance check.

To give the reader an example of the number of clauses processed by one run, we give figures obtained by a 270 seconds run of Vampire on the problem LCL-129-1.p of the PTTP library [Sutcliffe and Suttner 1998] using a 230MHz SPARC processor. During the run 8,272,207 clauses were generated, of which 5,203,928 were not included in the search space because their weights exceeded the specified weight limit. Of the remaining 3,068,279 clauses, at the end of the run 8,053 were retained, and 3,060,226 were rejected by *forward subsumption* (i.e., identified as instances of other clauses in the search space). Even if we assume that all the runtime was spent for checking subsumption, this means that Vampire was performing, on the average, over 10,000 subsumption-checks per second, each of these checks identifies if a clause is an instance of several thousand other clauses in the search space. An example term of weight 16 (i.e., with 16 symbols) participating in the proof is $t(e(e(x_0, e(x_1, x_2)), e(e(x_0, e(x_3, x_2)), e(x_1, x_3))))$, but most clauses had the weights between 20 and 24, and some of them 40. Doing such a number of subsumption-checks per second without using term indexing is hardly possible.

Tableau-based provers. The most known tableau-based prover is SETHEO [Letz, Schumann, Bayerl and Bibel 1992, Moser et al. 1997]. Such provers start with a set of input clauses, and construct a tree-like proof object, whose nodes are literals or clauses. At each step of the algorithm, one selects a node in the tree and applies some inference rule that produces a set of children of the selected node. Usually, this inference rule is resolution against one of the input clauses, and the set of the input clauses does not change during the run.

This can be implemented in the same way as SLD-resolution rule is implemented in logic programming (as observed by Stickel [1988]), because every input clause $A_1 \vee \dots A_n$ can be regarded as n Prolog clauses, each one of the form

$$A_i \texttt{ :- } \neg A_1, \dots, \neg A_{i-1}, \neg A_{i+1}, \dots, \neg A_n.$$

Another commonly exploited inference rule is *resolution against a lemma*, i.e., a previously derived literal. The set of lemmas is dynamically changing and can be large. In addition to these rules, backward and forward subsumption on the set of lemmas can be performed.

The performance of the tableau-based provers can drastically improve if all the main operations (resolutions against an input clause and against a lemma; and subsumption on lemmas) are implemented using term indexing.

2.2.2. Logic programming and deductive databases

In *logic programming languages* such as Prolog and *deductive databases*, a program is defined by a sequence of *clauses*. The evaluation of such programs may proceed in either a top-down fashion, similar in operation to tableau-based provers, or in a bottom-up fashion, which is similar to saturation-based provers.

In top-down evaluation, the evaluator identifies the substitutions which unify a literal in the goal clause with the heads of the program clauses. After this, the literal is replaced by the terms in the body of a unifying clause. Thus, the indexing problem in this context is based on unifiability of a term with one of the clause heads. When a subgoal is ground, we can make use of term matching, which is more efficient than unification. (Groundness can often be deduced using a procedure called *mode analysis*.) When goals are known to be ground, the indexing problem is one of identifying clause heads that are generalizations of the subgoals.

Tabled logic programming combines the goal-directed nature of top-down evaluation with the stronger completeness properties of bottom-up evaluation. Operationally, a tabled evaluator operates like a top-down evaluator, but remembers the results of previous evaluation to eliminate repeated computations involving any one subgoal. Thus, in addition to the problem of identifying unifiable clause heads, we are also interested in identifying if a new subgoal has been encountered earlier. In the simplest case, this may be determined by checking if the new goal is a *variant* of a previously encountered subgoal. In a more general case, we are interested in identifying subgoals that are *subsumed* by previously evaluated subgoals.

2.2.3. Functional programming and term rewriting

In *functional programming* and *term rewriting*, we are interested in computing the *normal form* of a term t with respect to a set of rewrite rules. Specifically, we identify a rewrite rule $l \to r$ such that its left-hand side l is a generalization of a subterm t' of t. We then replace t' in t by the corresponding instance of r. This process is repeated until we obtain a term t'' that contains no instances of any left-hand side. Thus, one of the main indexing problem of interest here is that of retrieving terms that are generalizations of a given term. Moreover, it is important to develop indexing strategies that can select not only candidate left-hand sides of rewrite rules, but also subterms within the input term where reductions are to be applied.

2.2.4. Summary of retrieval problems

In summary, we are interested in the following retrieval problems:

- atoms *unifiable* with a given atom (logic programming, deductive databases, different resolution rules);
- subgoals that are *variants* of previously attempted subgoals (tabled logic programming and deductive databases) [Chen and Warren 1996, Ramakrishnan 1991], naming in splitting without backtracking [Riazanov and Voronkov 2001*b*];
- atoms that are *instances* of previously computed atoms (tabled logic programming, deductive databases [Rao, Ramakrishnan and Ramakrishnan 1996], forward subsumption and demodulation in resolution-based theorem provers);
- atoms that are *generalizations* of previously computed or stored atoms (optimization in logic programming when predicates are known to be called with bound arguments, backward subsumption and demodulation in resolution-based theorem provers).

For some of these retrieval problems, their multiliteral analogues exists (e.g., simultaneous unifiability, forward and backward subsumption, and the clause variance problem). This leads to *multiterm indexing* discussed in Section 13.

2.3. Term indexing operations

Retrieval of candidate terms. Given a query term t and the indexed set $\mathcal{L}$, the retrieval operation is concerned with the identification of the subset $\mathcal{M}$ of those terms in $\mathcal{L}$ that have the specified relation R to t. The retrieval relation R identifies those terms $l \in \mathcal{L}$ that need to be selected. Some of the retrieval conditions discussed above are:

$$\begin{aligned}
\mathit{unif}(l,t) &\quad\Leftrightarrow\quad \exists\sigma\; l\sigma = t\sigma;\\
\mathit{inst}(l,t) &\quad\Leftrightarrow\quad \exists\sigma\; l = t\sigma;\\
\mathit{gen}(l,t) &\quad\Leftrightarrow\quad \exists\sigma\; l\sigma = t;\\
\mathit{var}(l,t) &\quad\Leftrightarrow\quad \exists\sigma\; (l\sigma = t \wedge \sigma \text{ is a renaming substitution}).
\end{aligned}$$

More complex conditions specific to indexing on multiliteral clauses will be introduced later.

Index construction and maintenance. In order to support rapid retrieval of candidate terms, we need to process the indexed set into a data structure called the *index*. *Index construction* operation is concerned with the initial construction of this data structure for a given operation R and indexed set $\mathcal{L}$. After the initial construction, we may need to make changes to the indexed set via insertion or deletion of terms. *Index maintenance* operations start with an index for a set $\mathcal{L}$, and incrementally construct an index for another set $\mathcal{L}'$ that is obtained by insertion or deletion of terms in or from $\mathcal{L}$.

Different choices of indexing techniques typically reflect a different trade-off among the costs for performing each of the above three tasks (namely, retrieval, index construction, and index maintenance). For instance, in the context of functional and logic programming, the indexed set is essentially fixed and so there are no index maintenance operations. Moreover, the index is constructed at compile-time. Thus the indexing techniques are aimed at optimizing the retrieval time, possibly at the expense of increasing the cost of index construction and maintenance. In some of the other applications such as tabled logic programming, insertions in the index may be frequent, but deletions do not occur. In other applications, such as automated theorem proving, the indexed set is generated at runtime and/or changes frequently, so it is necessary to minimize the costs of all three tasks.

2.4. Variations in term indexing operations

Nonlinearity. In general, the query and indexed terms may be *nonlinear*, i.e., may have repeated occurrences of variables. The multiple occurrences may occur all within one of the two terms involved, or a single variable may occur in both the query term and an indexed term. In either case, it is necessary to check the consistency among the substitutions received by multiple occurrences of the same variable. For instance, consider the problem of determining if the term $f(t_1, t_2)$ is an instance of $f(x, x)$. In this case, the first occurrence of x gets t_1 as its matching substitution, whereas the second occurrence of x gets t_2 as the substitution. In order for $f(t_1, t_2)$ to be an instance of $f(x, x)$, these two substitutions for x must be consistent, i.e., t_1 must be the same as t_2. Consistency checking is typically an expensive operation, and unless treated very carefully, consideration of nonlinearity at the indexing stage can lead to performance degradation. As such, many techniques ignore nonlinearity at the indexing stage, and rely on a post-processing step to carry out the consistency checks. Other methods postpone all of the consistency checking operations after the less expensive operations that merely check local term structure. For these reasons, we will deal mainly with linear terms towards the beginning of this paper. We will moreover assume that no variables are shared between the query term and indexed terms.

Equational theories. The retrieval condition may be based on an *equational theory* E, e.g., whether the query term t is an instance of a term from $\mathcal{L}$ with respect to E, i.e., there exists a term $l \in \mathcal{L}$ and a substitution β such that $E \vdash l\beta = t$. For instance, in the context of automated reasoning, we are interested in matching and unification in the presence of associative-commutative operators. In lazy functional programming, we are interested in matching in the context of the equational theory given by the program.

Priorities. In many contexts, the terms in the indexed set are associated with *priorities*, which need to be respected by the retrieval operation. In some contexts, such as functional programming, we may be interested in retrieving only the highest priority patterns, i.e., such terms l that

$$R(l,t) \wedge \forall l' \in \mathcal{L} \ \ (\mathit{priority}(l') > \mathit{priority}(l) \Rightarrow \neg R(l',t)).$$

In other contexts such as logic programming and deductive databases, the retrieval operation may be required to return candidate terms from $\mathcal{L}$ in a decreasing order of priority. Priorities arise in the context of automated reasoning as well, where they may be used to encode heuristics aimed at generating "simpler," "more general" or "more useful" theorems first. For instance, in completion procedures we may prefer to generate critical pairs from "smaller" terms before using "larger" terms.

Computing substitutions. In many applications, we are not only interested in identifying the candidate terms from the indexed set, but also in identifying a substitution or substitutions under which the query term and the candidate term satisfy the retrieval condition. In theory, computation of such substitutions can be performed after indexing, but this approach increases the post-processing cost after identification of the candidate terms. As such, many indexing techniques are designed to compute the substitutions as part of the indexing operation and return them. The *one-at-a-time* retrieval mode is particularly suited for such techniques.

Many-to-many operations. Many-to-many indexing problems arise in the context of operations that are performed collectively on groups of terms. Some of the most common examples are multiliteral subsumption, hyperresolution, and unit-resulting resolution. Substantial speedups have been reported by using many-to-many indexing operations in these cases, as compared to just using one-to-many indexing. In particular, the many-to-many problems require that we deal with a set $\mathcal{Q}$ of query terms, rather than a single query term.

Since the term "many-to-many" is ambiguous and may refer to an operation on two indexes, while all the above mentioned operations require one indexed set of terms and several query terms, we will use the term *multiterm* indexing. We postpone the discussion of multiterm indexing until Section 13.

Subterm-based retrieval conditions. Sometimes, we may be interested in considering all subterms of the query term as query terms themselves. For instance, in term rewriting, we want to identify an indexed term l (which corresponds to the lhs of a rewrite rule) and a *subterm* t/p of the given term such that t/p is an instance of l.

Similarly, we may want to index on all of the subterms of a given set of indexed terms. For example, in *completion procedures* or *paramodulation-based theorem proving* (see [Nieuwenhuis and Rubio 2001], Chapter 7 of this Handbook) we have to identify all subterms of given terms that are unifiable with, or instances of, a query term.

Although all of these cases can be handled either by multiterm indexing or by a straightforward inclusion of all subterms in the index, it would be advantageous to exploit the fact that a term and its subterms share common structures in order to develop space and time-efficient indexing algorithms.

Consideration of subterm-based retrieval conditions is beyond the scope of this paper.

3. Data structures for representing terms and indexes

3.1. Data structures for representing terms

There are several data structures for representing terms. We present a short summary of three such structures: conventional representations using trees or DAGs, flatterm representation and Prolog term representation.

3.1.1. Conventional representation of terms

Given the inductive definition of terms, the most obvious way to represent them is to use trees or *directed acyclic graphs* (*DAGs*). The root node of the tree representation for a term t contains the root symbol of t and pointers to nodes that correspond to immediate subterms of t. These pointers may be stored either in an array or as a linked list. The conventional representation is a versatile one and readily supports common operations such as traversing a term in different ways, skipping subterms, etc.

An example of a DAG representation is shown in Figure 1. As compared to a tree representation, a DAG representation presents an opportunity to *share subterms*. Such sharing can contribute to as much as an exponential decrease in the size of the term. The benefits of sharing are significant in practice — for instance, in term rewriting and functional programming, rules such as $f(x) \to g(x,x)$ occur commonly, and a tree-representation would require duplication of the (arbitrarily large) term that appears as the substitution for x in the term being reduced. In contrast, using a dag representation, we can achieve the same effect by duplicating the pointer to the substitution, without having to duplicate the substitution.

Some systems employ dag representations with *aggressive sharing*, also called *perfect sharing* where we ensure that only a single copy of a term exists, regardless of the number of contexts in which it occurs. Aggressive sharing is used in some the-

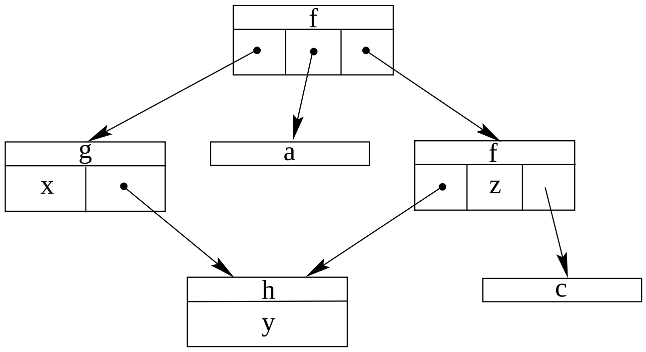


Figure 1: DAG representation of the term $f(g(x, h(y)), a, f(h(y), z, c))$

orem provers, such as OTTER and VAMPIRE, especially for long-lived terms (terms that are *kept*). Aggressive sharing also simplifies nonlinear matching, since the task of consistency checking across multiple substitutions for the same variable simplifies to comparing the pointers to all the respective terms, rather than checking whether the substitutions are structurally identical. Particularly efficient representations have been developed for such aggressive sharing in the context of the congruence closure problem [Nelson and Oppen 1980]. The overhead of aggressive sharing is typically too high in programming applications such as functional and logic programming systems. However, in some contexts, a further optimized representation known as *hashed cons* has been found to be useful.

3.1.2. Flatterms

Flatterm is a representation for terms introduced in [Christian 1989, Christian 1993]. Flatterm is a linear structure that corresponds to a linked-list representation of the nodes visited in a preorder traversal of a term. In this paper, we will use *preorder traversal* as synonym for depth-first, left-to-right inspection of the subterms. To facilitate skipping of subterms, a node n corresponding to a subterm has pointers to the node that follows immediately after all of the children of n. The flatterm representation of the term $f(g(a, h(y)), h(y), x)$ is shown below, where the following notations are used:

Symbol: symbol at the current node
Next: next node in preorder traversal of term
Prev: previous node in preorder traversal
End: last node (in preorder traversal) in the subtree rooted at the current node.

Observe that most operations on terms require some form of traversal of a term. If we restrict ourselves to a preorder traversal, then flatterm provides a more efficient way to traverse the term as compared to conventional terms. Moreover, each node in a flatterm has a fixed structure with the same number of fields. This implies that the sizes of all nodes are identical, which, in turn, simplifies memory management.

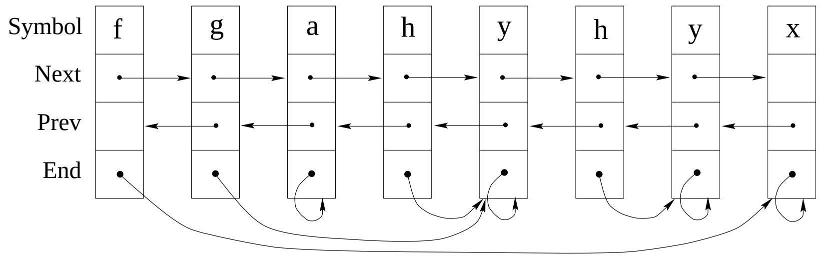


Figure 2: Flatterm representation of $f(g(a,h(y)),h(y),x)$

(In contrast, the size of a node in a conventional term depends on the number of children of the node.) The constant size of the nodes means also that they can be put in an array, which eliminates the need in the **Next** and **Prev** references and leads to a smaller memory consumption and faster traversal.

Flatterms are particularly efficient when used in conjunction with left-to-right discrimination trees (see Section 6). They are not well-suited in situations where the traversal order may be different from left-to-right. Another difficulty is that this representation does not support structure sharing. Flatterms are used in several theorem provers to represent query terms, for which structure is unimportant.

3.1.3. Prolog terms

Prolog uses an optimized version of the conventional term representation, where each node is tagged as a *const*, *fun* or *ref* node, which are used to store constants (i.e., function symbols with arity zero), function symbols (with arity > 0), and pointers to other terms respectively. The variable nodes are represented as references to the node itself, which enables particularly efficient implementation of variable binding via setting of the pointer to the term to be bounded. Note that neither the var nodes, nor the const nodes have any children. Hence Prolog implementations store such values within the corresponding parent nodes. Finally, the function nodes are represented using variable-sized nodes, with both the function symbol and references to the children represented using the same array. The Prolog representation of terms is illustrated in Figure 3.

3.2. Variable banks

There are situation where the same term may be used with different sets of variables during the same retrieval operation. For example, the (indexed) literal $f(y,g(y))$ can be used n times for performing a hyperresolution inference with the clause

$$\neg f(x_0,x_1) \vee \neg f(x_1,x_2) \vee \neg f(x_{n-1},x_n) \vee h(x_0,x_n).$$

To perform this hyperresolution inference, one needs to use n copies of $f(y,g(y))$

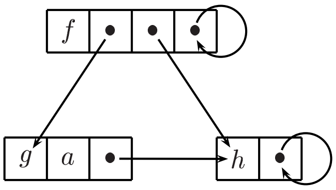


Figure 3: Prolog representation of $f(g(a, h(y)), h(y), x)$

with y instantiated by the terms $x_0, g(x_0), \ldots, g^{n-1}(x_0)$.

A typical implementation technique for such situations is the use of *variable banks*. Suppose that we have an indexed term l with the variables $y_1, \ldots, y_n$ which can be used several times with different instantiations for $y_1, \ldots, y_n$. Then we create several copies of the sequence $y_1, \ldots, y_n$:

$$\begin{array}{ccc} y_{11}, & \cdots & , y_{1n}, \\ & \cdots & \\ y_{m1}, & \cdots & , y_{mn}. \end{array}$$

Each copy is called a variable bank. When, during retrieval, we perform kth operation with the index, we use the variable bank $y_{k1}, \ldots, y_{kn}$ instead of $y_1, \ldots, y_n$. To increase efficiency, variable banks are usually allocated only once as an array, and each variable bank is also implemented as an array containing a large enough number of variables.

Variable banks are used (under different names) in Otter, Fiesta, and Vampire. The term "variable banks" is taken over from [Rivero 2000].

3.3. Data structures for representing indexes

There are two broad categories of representations for indexes. The first category consists of representations similar to finite-state automata or tries, which are "interpreted" at retrieval time. The second category consists of representations that use some form of code that is executed in order to perform retrieval.

3.3.1. Automata-based representations

The *automata-like representations* deal with issues very similar to those that arise in *string-matching automata*, and these issues have been studied well in the literature. Perhaps the most important aspect is the representation for outward transitions from a state. A variety of techniques have been studied that attempt to minimize the storage needed for representing these transitions, yet try to achieve $O(1)$ expected time for identifying the applicable transition based on the symbol in the input string (or term). The different data structures studied are:

- *Array:* fast, but high memory consumption since the number of outward transitions may be very small compared to the alphabet size.
- *Linked list*: economical in terms of space, but slow for making transitions
- *Hash table:* storage requirements slightly more than linked list representation, but significantly faster. However, collisions can become a problem.
- *Jump table:* specialized data-structures that combine the benefits of array representation with low memory requirements [Dawson, Ramakrishnan and Ramakrishnan 1995].

In jump tables, the symbol's value is directly used as an index, as in the case of the array representation. However, to reduce the storage requirements, the tables for different automaton states are "overlapped." The problem of optimizing the space requirement is NP-complete, effective heuristics have been developed in [Dawson, Ramakrishnan and Ramakrishnan 1995]. These techniques provide almost 100% space efficiency, and represent a significant improvement over earlier techniques.

3.3.2. Code trees

It is usual in automated deduction to compile the query term into a code that is executed on the index. If the compilation time is small compared to the retrieval time, this compilation pays off.

It is less usual to compile the index itself to a code that will be executed on the query term in order to perform retrieval. We may compile the code for a real machine, as for functional programming language compilers, or for a virtual machine, as in the case of Prolog's WAM (see, e.g., [Aït-Kaci 1991]). In the case of automated deduction, the dynamic nature of indexed sets and the modern computer architecture work against a fully compiled approach, so a virtual machine is the only alternative. Such an approach to indexing is demonstrated in [Voronkov 1994, Voronkov 1995, Riazanov and Voronkov 2000*b*], where the index is represented as a structure called *code tree*. Code trees will be discussed in Section 9.

4. A common framework for indexing

Symbol-based indexing techniques can be described abstractly as follows. Candidate terms selected are those that have identical function symbols as the (subset of) positions in the query and the indexed terms. In order to select the candidate terms, we need to examine these subset of positions in the query term in some order. In effect, the whole process can be viewed as constructing a string of symbols from the query term, and identify if this string "matches" the string constructed from the indexed terms. Thus, most symbol-based indexing techniques can be unified under the broad theme of *string matching based indexing*.

Given this analogy between string matching and term indexing, we can readily see the applicability of previously known techniques for string-matching, such as the use of trie and finite-state automata based data structures for fast matching. Specifically, we can build a trie or an automaton consisting of all the strings obtained from the indexed terms. At retrieval time, a string corresponding to the query term is constructed (implicitly or explicitly) and "run through" the automaton to identify the strings in the automaton that are compatible with this string. Finally, the compatible strings in the automaton are mapped into the corresponding candidate set of terms. The key advantage of using an automaton (or a trie) representation is that those operations that are common for matching against multiple strings can be "factored out." This advantage results in substantial speedups over a naive approach that would repeatedly test the query term against each one of the indexed term.

In this section we first develop the concepts needed to relate string matching operations and the corresponding relations among terms. We then proceed to describe the basic techniques, many of which are drawn from string-matching, that can be used to improve the speed of retrieval and/or reduce the size of the index. Our description of indexing techniques in subsequent sections will draw upon the concepts developed in this section. This approach enables one to understand many known indexing methods as instances of a more generic set of techniques, thus shedding light on the fundamental advantages and disadvantages of each of these methods and making it easier to compare and contrast them.

4.1. Position strings

In order to make use of string-matching techniques, we first need to convert the terms into a string representation. One way to do this is to write out the symbols occurring in a term in some sequence, thus arriving at a string. However, such an approach may lose some of the information captured by the term structure. To preserve this information, we can capture each symbol in a term together with its position in the string. For instance, we can represent the term $f(a, g(b, c))$ as a string

$$\langle \Lambda, f\rangle\langle 1, a\rangle\langle 2, g\rangle\langle 2.1, b\rangle\langle 2.2, c\rangle. \tag{4.1}$$

Rather than generating a single string from a term, we may choose to generate multiple strings. For instance, we may generate the following set of strings from the same term:

$$\{\langle \Lambda, f\rangle\langle 1, a\rangle, \langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.1, b\rangle, \langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.2, c\rangle\}. \tag{4.2}$$

We refer to strings (4.1) and those in (4.2) as *position strings*, or *p-strings* for short. Intuitively, a p-string is simply a string representation of some term. More formally,

4.1. Definition *(Position strings).* A *position string* (abbreviated *p-string*) S over an alphabet Σ is a nonempty string of the form $\langle p_1, s_1\rangle\langle p_2, s_2\rangle \cdots \langle p_n, s_n\rangle$ where $p_i \in \mathcal{P}$ and $s_i \in \Sigma \cup \mathcal{V}$ such that:

- for all $1 \leq i, j \leq n$, if p_i is a proper prefix of p_j then $i < j$;
- there exists a term t, called the *characteristic term for S* such that
  - for every $1 \leq i \leq n$ we have $root(t/p_i) = s_i$; and
  - $p_1, \ldots, p_n$ are all positions in t.

Intuitively, we can view the positions $p_1, \ldots, p_n$ in a p-string as capturing a way to traverse a term, with $s_1, \ldots, s_n$ being the symbols visited in this traversal order. If the order of traversal is fixed *a priori* (e.g., we use a depth-first or breadth-first traversal order), then the position information becomes redundant. If we do not want to constrain ourselves with any one fixed way of visiting symbols in a term, then the position information is important.

In a term structure, it is typically meaningless and/or impossible to visit a node before first visiting all of its parents. This is the reason for imposing the first condition in the above definition. Secondly, we want the p-string to represent a term, which we call the characteristic term. Note that the characteristic term is unique, up to replacement of variables by other variables.

Sometimes, we may abbreviate p-strings in such a way that we drop one or more variables from them. For instance, we abbreviate the p-string

$$\langle \Lambda, f\rangle\langle 1, *\rangle\langle 2, g\rangle\langle 2.1, *\rangle\langle 2.2, c\rangle$$

as $\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.2, c\rangle$. We will use the notation $ct(S)$ to denote the characteristic term of a string S. The characteristic term for the above p-strings is $f(*, g(*, c))$.

4.2. P-string compatibility and indexing

We now proceed to describe how p-strings generated from the indexed terms can be used as the basis for identifying those terms that are compatible with a given query term. For this purpose, we need to extend the notion of R-compatibility to operate between p-strings and terms:

4.2. DEFINITION *(p-string compatibility).* Given a term t and a p-string S, define the *S-prefix* of t, denoted $t \backslash S$, to be a term t' obtained from t by replacing every position in t that is not contained in S by a new distinct variable. S is said to be *R-compatible* with the term t if $R(ct(S), t')$ holds.

We will overload the notation $R(S, t)$ to denote R-compatibility between the string S and term t. For instance, the p-string $\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.2, c\rangle$, whose characteristic term is $f(*, g(*, c))$, is compatible with the query term $f(a, g(b, x))$ with respect to the retrieval condition *unif*. On the other hand, the p-string $\langle \Lambda, f\rangle\langle 1, c\rangle\langle 2, g\rangle\langle 2.2, c\rangle$ is not *unif*-compatible with this term. Moreover, $\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.2, c\rangle$ is not compatible with this term with respect to the relation *gen*.

A simple technique for determining if $R(l, t)$ can possibly hold is to first generate one or more p-strings from l and then check if each of these p-strings are R-compatible with t. To ensure that an indexing technique based on this approach will be *sound* (i.e., identify all indexed terms that are potentially R-compatible with t), we require that these p-strings be *characteristic strings* of l as defined below:

4.3. Definition *(Characteristic set of strings).* A set $\{S_1, \ldots, S_k\}$ of p-strings is called *characteristic for a term* l if for any term t, we have $R(l,t) \Rightarrow \bigwedge_{1 \le i \le k} R(S_i, t)$.

This condition ensures that any term that is potentially compatible with t is identified by the filter. We will use the notation $\mathcal{S}_t$ to denote a characteristic set of strings for a term t. If this set is singleton, then we use $\mathcal{S}_t$ to refer to this string. For a set $\mathcal{L}$ of terms, we use the notation $\mathcal{S}_\mathcal{L}$ to refer to the union of characteristic sets for all terms in t, i.e., $\mathcal{S}_\mathcal{L}$ denotes a set $\bigcup_{l \in \mathcal{L}} \mathcal{S}_l$.

Symbol-based indexing techniques are based on constructing a characteristic set $\mathcal{S}_\mathcal{L}$ of strings for the indexed set $\mathcal{L}$ of terms, and constructing an automaton (or trie) of all these strings. Such an automaton can be used to quickly identify those p-strings that are compatible with a given query term. This information about compatibility with individual p-strings needs to be combined to identify which of the indexed terms are potentially compatible with the query term. In particular, we consider each indexed term l, and ask the question if the query term was compatible with strings of its characteristic set. In the worst case, the combination step will hence take time $O(\sum_{l \in \mathcal{L}} |\mathcal{S}_l|)$. However, in practice, several additional constraints are placed upon the traversal orders for generating characteristic strings, which enable the indexing to be performed faster.

To illustrate the concepts developed so far, consider the indexed set $\mathcal{L} = \{f(*, a, b), f(b, a, a), f(b, a, *)\}$ under the retrieval condition *gen*. Suppose that we generate one characteristic string from each of the terms in the indexed set, so that we get

$$\mathcal{S}_\mathcal{L} = \{\langle \Lambda, f\rangle\langle 2, a\rangle\langle 3, b\rangle, \langle \Lambda, f\rangle\langle 2, a\rangle\langle 3, a\rangle\langle 1, b\rangle, \langle \Lambda, f\rangle\langle 2, a\rangle\langle 3, *\rangle\langle 1, b\rangle\}.$$

An automaton for these three strings is shown in Figure 4(a). In this figure, the leaves are annotated with the number(s) of term(s) from which the p-string corresponding to the state was generated.

Given a query term $t = f(b, a, c)$ and the retrieval condition *gen*, we can use this automaton for indexing as follows. First, we inspect the position Λ of t, note that the symbol f at this position is identical to that on the transition from state 2 to state 3, and thus move to state 3 of the automaton. Next, we inspect position 2 of t, match with the symbol a on the transition leading to state 5. Next we inspect position 3, and note that it is compatible with $*$ on the transition leading to state 8. Finally, we inspect position 1 and then reach the final state 13 that is marked with $\{3\}$. Since the query term is compatible with only the p-string derived from the indexed term $f(b, a, *)$, we can immediately conclude that the query term is potentially compatible with this term alone.

Backtracking may be needed while performing retrieval operation in some cases. For instance, for the query term $f(b, a, b)$, we can use the automaton of Figure 4(a) to reach the final state marked with $\{1\}$. However, this term is compatible with the

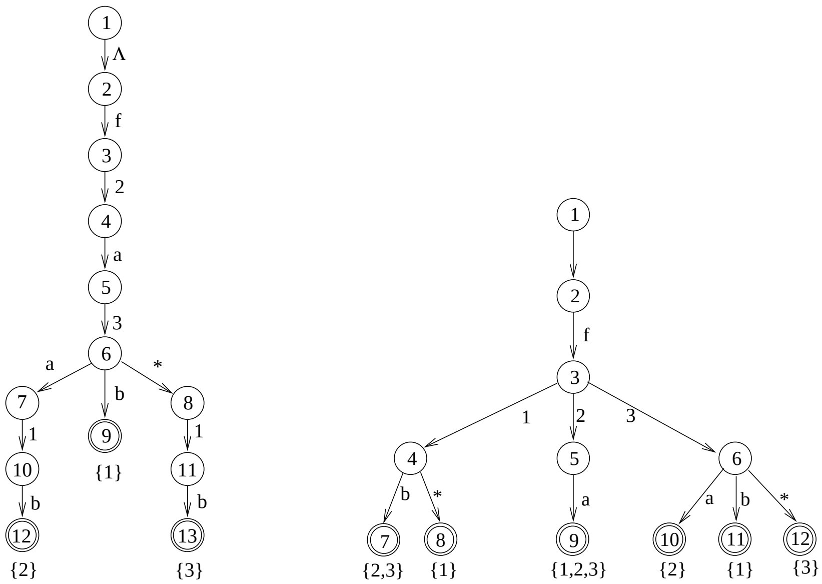


Figure 4: Example indexing automata

p-string generated from the term $f(b, a, *)$ as well, and this can be detected only by backtracking to state 6 and following down the transition to state 8.

Now consider the same set of indexed terms, but with different sets of p-strings and the retrieval condition *unif*. Let the p-strings be:

$$\begin{aligned}
\mathcal{S}_{f(*,a,b)} &= \{\langle\Lambda, f\rangle\langle 1, *\rangle, \langle\Lambda, f\rangle\langle 2, a\rangle, \langle\Lambda, f\rangle\langle 3, b\rangle\},\\
\mathcal{S}_{f(b,a,a)} &= \{\langle\Lambda, f\rangle\langle 1, b\rangle, \langle\Lambda, f\rangle\langle 2, a\rangle, \langle\Lambda, f\rangle\langle 3, a\rangle\},\\
\mathcal{S}_{f(b,a,*)} &= \{\langle\Lambda, f\rangle\langle 1, b\rangle, \langle\Lambda, f\rangle\langle 2, a\rangle, \langle\Lambda, f\rangle\langle 3, *\rangle\}.
\end{aligned}$$

Note that there is one p-string corresponding to each of the root-to-leaf paths in each indexed term. The automaton for these p-strings, shown in Figure 4(b), thus captures the *path indexing* technique proposed in [Stickel 1989, Ramesh, Ramakrishnan and Warren 1990] and discussed in the next section. We can use this automaton for retrieving terms that are unifiable with the query term $t = f(b, *, b)$. We need to retrieve the indexed terms such that all of the p-strings generated from them are compatible with t. We use the automaton to inspect t. We match the root symbol f, and then proceed to follow down on each of the transitions down from state 3. Specifically, we inspect position 1 in t at state 4, and see that it is compatible (under unification) with both transitions leading out of state 4. Thus, t is compatible with the first p-strings of all of the indexed terms. We now match against the second strings by following the second transition out of state 3. Again, we see that under unification, t is compatible with all of these p-strings as well.

Finally, we follow the third transition out of state 3, and find that states 11 and 12 are compatible with t. At this point, we have information about the compatibility of t with all of the p-strings, from which we can conclude that t is potentially compatible with terms 1 and 3.

5. Path indexing

5.1. Overview of path indexing

We first describe *path indexing* using the general scheme presented in Section 4. In path indexing, we generate multiple p-strings from each indexed term, each one corresponding to a traversal of one root-to-leaf path in the term. We then build a trie of these p-strings, which is used to perform the retrieval operation.

By exploiting the nature of root-to-leaf traversals, we can use a more optimized representation for positions. In particular, since successive positions inspected correspond to parent and child, it is sufficient to denote which child is being included in a p-string, rather than specifying the position of the child. For instance, consider the path from the root of the term $f(b, g(f(x), a))$ to the variable x. Rather than using a p-string $\langle \Lambda, f\rangle\langle 2, g\rangle\langle 2.1, f\rangle\langle 2.1.1, *\rangle$, we can use the simpler representation $f.2.g.1.f.1.*$. For the rest of our discussion on path indexing, we will make use of the simplified representation for p-strings. We will also use the alternative term *path strings* to refer to such p-strings.

The second optimization is in the way we combine the results of matches on p-strings into a match for (some of) the indexed terms using set intersection operations. We illustrate this using the example set of indexed terms

$$(1)\ f(g(a,*),c),\quad (2)\ f(g(*,b),*),\quad (3)\ f(g(a,b),c),$$
$$(4)\ f(g(*,c),b),\quad (5)\ f(*,*).$$

The path strings generated from these terms, together with an identification of which indexed terms produced the path strings, are shown below:

$f.1.*$	$\{5\}$	$f.2.*$	$\{2,5\}$
$f.1.g.1.*$	$\{2,4\}$	$f.2.c$	$\{1,3\}$
$f.1.g.1.a$	$\{1,3\}$	$f.2.b$	$\{4\}$
$f.1.g.2.*$	$\{1\}$		
$f.1.g.2.b$	$\{2,3\}$		
$f.1.g.2.c$	$\{4\}$		

A trie for these strings is shown in Figure 5. We have annotated each leaf node in this trie with the set of indexed terms that generated the path string corresponding to the leaf.

To combine the results of matching individual path strings, we can proceed as follows. First, consider each state s with outward transitions on positions $q_1, \ldots, q_n$.

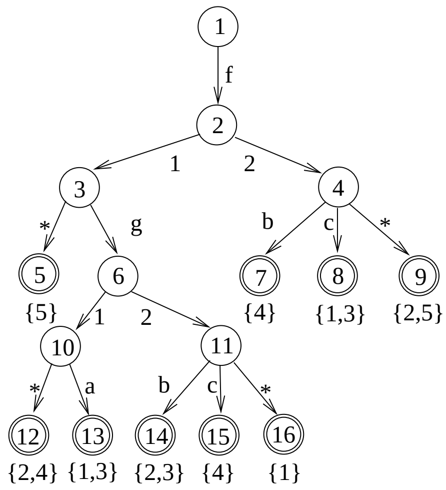


Figure 5: Path index for $\{f(g(a,*),c), f(g(*,b),*), f(g(a,b),c), f(g(*,c),b), f(*,*)\}$

Observing that the path strings are generated using a consistent traversal order, each indexed term that is compatible with the path string prefix inspected up to s would have generated one path string corresponding to inspecting each of these positions $q_1, \ldots, q_n$. Thus, we can use set intersections to combine the indexed terms that are identified as being compatible with the query term along each one of these transitions. Now consider a state s that has an outward transition on symbols $a_1, \ldots, a_n$ and $*$. Note that while retrieving generalizations, any query term that is compatible with a path string $s_1.p_1.s_2.p_2 \ldots .p_k.a_j \ldots .$ will also be compatible with the path strings $s_1.p_1.s_2.p_2 \ldots .p_k.*$. Consequently, at retrieval time, we need to take the union of the candidate set corresponding to the state reached by following the $*$ transition with the candidate set identified by following the transition corresponding to the nonvariable symbol that appears at this position in the query term. By employing set union and intersection operations, we can thus combine the results of matching individual path strings efficiently, and moreover, do so without any backtracking.

Some of the earliest ideas related to path indexing can be found in the coordinate indexing scheme [Hewitt 1971]. Path indexing was proposed independently by Ramesh et al. [1990] in the context of logic programming, and Stickel [1989] in the context of automated reasoning. Path indexing and its many variants have been extensively studied by McCune [1992] and Graf [1992, 1996].

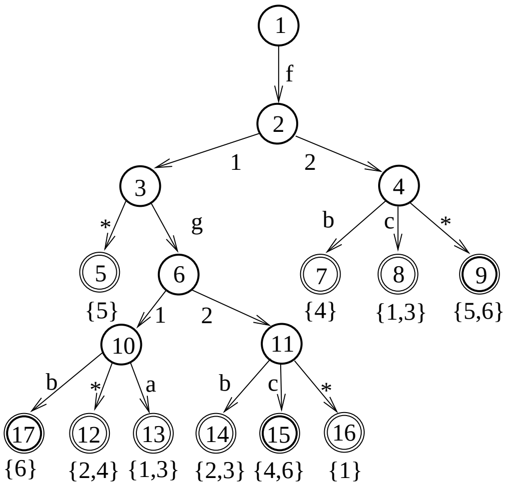


Figure 6: Path index after insertion of $f(g(b,c),*)$ to Figure 5

5.2. Indexing algorithms

The construction and maintenance operations involve insertion and deletion of terms from the index. Below we describe algorithms for insertion, deletion, and retrieval.

5.2.1. Index construction

Index construction proceeds by successive insertion of path strings from each of the indexed terms. The insertion process is straightforward. We generate path strings corresponding to each of the root-to-leaf paths in the term to be inserted. For those path strings that already appear in the index, we simply need to insert the new indexed term in the candidate set associated with the final state corresponding to the string. Those path strings that do not already exist in the trie are inserted into the trie, and the final state corresponding to the string is annotated with the singleton set containing the newly inserted indexed term. Figure 6 shows the result of inserting the term $f(g(b,c),*)$ in the index of Figure 5. Note that two of the path strings generated from the term, namely, $f.1.g.2.c$ and $f.2.*$, already exist in the trie. The candidate sets of the corresponding final states (labelled 15 and 9 respectively) are updated to include the newly included term, which is identified by the number 6. The third path string, $f.1.g.1.b$ is a new path string, and it is inserted into the trie. The corresponding new final state, labelled 17, is associated with the candidate set $\{6\}$.

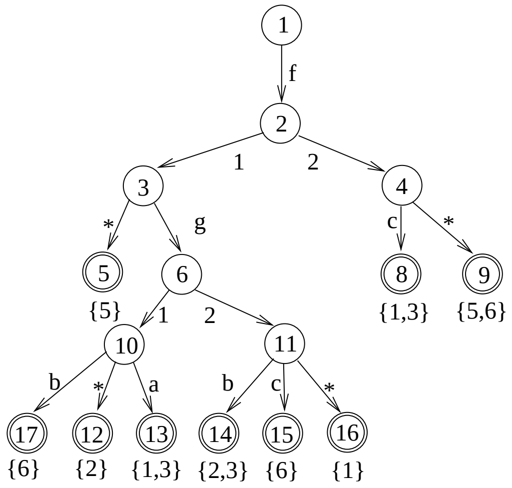


Figure 7: Path index after deleting $f(g(*,c),b)$ from Figure 6

5.2.2. Index maintenance

Index maintenance involves insertion and deletion of terms from the indexed terms. Insertion of new terms has already been covered. Deletion of terms is also simple. We first generate the path strings from the term to be deleted. For each of these strings, we identify the corresponding final state in the trie and delete the indexed term from the associated candidate set. If this operation results in an empty candidate set, we can then delete the final state from the trie. If the parent of the final state has no children now, we can delete the parent as well and proceed higher up in the trie. This process is repeated until we reach a trie node that has nonzero number of descendants. The deletion operation is illustrated in Figure 7, which shows the index obtained by deleting the term $f(g(*,c),b)$ (which is identified by the number 4 in the indexed set) from the index of Figure 6. Note that this term generates three path strings $f.1.g.1.*$, $f.1.g.2.c$, and $f.2.b$, which are associated with final states labelled 12, 15, and 7 respectively. We delete 4 from the candidate sets associated with these states. Observe that this results in the candidate set of state 7 becoming empty. We therefore delete this state. Observe that the parent state 4 has other descendants, so we stop without deleting this state.

5.2.3. Retrieval of generalizations

Retrieval of generalizations corresponds to the one-to-many matching problem, where we are interested in rapid selection of terms such that the query term matches these terms. It is an important operation that plays a central role in term rewriting and functional programming. Even in the case of logic programming and deductive databases, one-to-many matching occurs frequently as an optimization of the unification operation when some arguments are known to be bound.

```
function retrieve(Trie state s, term u) returns set of terms M
    M := φ
    if u is nonvariable and ∃ a transition
        from state s to another state s' labelled with root(u) then
            if s' is a final state then
                M := M_{s'}
            else
                let I be the set of numbers i such that ∃
                a transition from s' to s_i labelled by i
                    M := ⋂_{i∈I} retrieve(s_i, u/i)
            endif
    endif
    if ∃ a transition from s to a state s_* labelled with * then
        M := M ∪ M_{s_*}
    return M
```

Figure 8: Algorithm for retrieval of generalizations from a path index

The retrieval process is best described using the algorithm of Figure 8. The algorithm takes two parameters, one representing a state within the trie, and the second a subterm of the query term t. It is initially invoked as $retrieve(s_0, t)$, where s_0 is the root of the trie. We use the notation $\mathcal{M}_s$ to denote the candidate set associated with a final state s. Also recall that $\mathcal{I}$ denotes the set of indexed terms.

We can perform one optimization to avoid set unions altogether at retrieval time. This can be accomplished as follows. Let s be a state in the trie that has a transition on $*$ to another state s'. Then we duplicate the indexed terms in $\mathcal{M}_{s_*}$ in the candidate sets associated with all the descendent states of s. This optimization reduces the cost of retrievals, but can increase the cost of insertions and deletions to the indexed set. When a new path string terminating with a $*$ is inserted in (or deleted from) the index, we have to not only update the candidate set associated with s_*, but also the candidate sets associated with all of the descendants of s. Even worse, when we delete a t from such an index, we have to check if some other terms having $*$ in nonvariable positions of t should be removed from the branches previously used by t.

5.2.4. Retrieval of instances

Retrieval of instances is an important problem that arises in the context of backward subsumption and demodulation in automated theorem-proving, and tabled resolution in logic programming and deductive databases. We describe a retrieval algorithm in Figure 9.

In identifying indexed terms that are instances of the query term, observe that variables in the indexed term play no role. (Once again, this is because we are

function $retrieve$(Trie state s, term u) returns set of terms $\mathcal{M}$
1. if $u = *$ then
2. let $\mathcal{F}$ be the set of all final states that are descendants of s
3. $\mathcal{M} = \bigcup_{s' \in \mathcal{F}} \mathcal{M}_{s'}$
4. else if $\exists$ a transition from s to a state s' labelled with $root(u)$ then
5. if s' is a final state then
6. $\mathcal{M} := \mathcal{M}_{s'}$
7. else
8. let $\mathcal{I}$ be the set of numbers i such that $\exists$ a transition from s' to s_i labelled by i
9. $\mathcal{M} := \bigcap_{i \in \mathcal{I}} retrieve(s_i, u/i)$
10. endif
11. endif
12. return $\mathcal{M}$

Figure 9: Algorithm for retrieval of instances from a path index

ignoring nonlinearity during indexing.) However, variables in the query terms are significant. In particular, if the query term has a path string of the form $s_1.p_2.s_2.p_3 \ldots .p_{k-1}.s_{k-1}.p_k.*$, then this path string is compatible with all path strings in the index of the form $s_1.p_2.s_2.p_3.\ldots.p_{k-1}.s_{k-1}\ldots$. Thus, we take the union of all the candidate sets corresponding to all such path strings at step 2 of the algorithm. If u is not a variable, then the retrieval proceeds as usual by taking the transition from s that is labelled with the root symbol of u.

Observe that the set $\mathcal{F}$ of the descendants of the state s identified at step 2 can be large, and as such, the union operation quite expensive. We can reduce this cost by precomputing the union and storing it at each state of the trie. Note that unlike the optimization for avoiding unions in the case of retrieval of generalizations, this optimization does not increase the cost of either insertion or deletion of indexed terms, but may considerably increase memory consumption. In particular, let $\mathcal{C}_s$ denote the set

$$\bigcup_{s' \in \mathcal{F}}$$

(see step 3 of the algorithm). Then, whenever a path string S is inserted into the trie, it is added to $\mathcal{C}_s$ for every state s that is on the root-to-leaf path in the trie corresponding to S. Deletion of indexed terms simply reverses this process.

5.2.5. Retrieval of unifiable terms

Retrieval of unifiable terms is an important operation in automated theorem proving in tasks such as resolution and critical-pair generation, and in logic programming and deductive databases. We present the algorithm for retrieving unifiable terms below.

function $retrieve$(Trie state s, term u) returns set of terms $\mathcal{M}$

1. if $u = *$ then
2.   let $\mathcal{F}$ be the set of all final states that are descendants of s
3. $\mathcal{M} = \bigcup_{s' \in \mathcal{F}} \mathcal{M}_{s'}$
4. else
5. $\mathcal{M} := \emptyset$
6.   if $\exists$ a transition from s to a state s' labelled with $root(u)$ then
7.   if s' is a final state then
8. $\mathcal{M} := \mathcal{M}_{s'}$
9.   else
10.   let $\mathcal{I}$ be the set of numbers i such that $\exists$ a transition from s' to s_i labelled by i
11. $\mathcal{M} := \bigcap_{i \in \mathcal{I}} retrieve(s_i, u/i)$
12.   endif
13. endif
14. if $\exists$ a transition from s to a state s_* labelled with $*$ then
15. $\mathcal{M} := \mathcal{M} \cup \mathcal{M}_{s_*}$
16. return $\mathcal{M}$

Figure 10: Algorithm for retrieval of unifiable terms from a path index

Observe that unification treats the indexed term and the query term symmetrically. In particular, if either term contains a variable at some position p, then all path strings that are identical in the other term up to p, are compatible with this path string. Thus, the retrieval algorithm for unification is obtained by essentially combining the retrieval algorithm for generalizations and instances. The combination is in some sense like a union, as the candidate sets obtained are the larger of the sets obtained at step 2 of Figure 9 and step 13 of Figure 8.

The optimizations mentioned earlier for avoiding the unions at steps 3 and 15 can again be applied, with essentially the same tradeoffs as in the case of retrieval of generalizations and instances.

5.2.6. Retrieval of variants

This operation is also symmetric with respect to the indexed and query terms. However, unlike unification, we treat variables in this case just like nonvariable symbols. The algorithm for retrieval is given in Figure 11.

5.2.7. Implementation issues

The set union and intersection operations can be made efficient by using bitvector representations for sets. This ensures that we can perform the unions and intersections very fast (e.g., in a few machine instructions) even for index sets with a few hundreds of terms. However, problems can arise when the sets contain

function $retrieve$(Trie state s, term u) returns set of terms $\mathcal{M}$
1. if $\exists$ a transition from state s to another state s' labelled with $root(u)$ then
2.   if s' is a final state
3.   then $\mathcal{M} := \mathcal{M}_{s'}$
4.   else
5.   let $\mathcal{I}$ be the set of numbers i such that $\exists$ a transition from s' to s_i labelled by i
6. $\mathcal{M} := \bigcap_{i \in \mathcal{I}} retrieve(s_i, u/i)$
7.   endif
8. endif
9. return $\mathcal{M}$

Figure 11: Algorithm for retrieval of variants from a path index

larger numbers of terms, for example over 20,000 as reported in [Riazanov and Voronkov 2001*a*].

5.3. Variations of path indexing

Use of bitvectors for compact representation of the sets $\mathcal{M}$ and $\mathcal{C}$ was suggested in [Ramesh et al. 1990]. It is especially appropriate in applications where the number of terms involved is small to moderate (up to a few hundred terms). If such compact and efficient representation was usable, further optimizations are possible. In particular, [Ramesh et al. 1990] suggests that we use the candidate terms identified so far (i.e., in indexing using the first k path strings in the query term) to prune the candidate set for subsequent path strings (i.e, $k+1$st path string). More precisely, we carry around the current candidate set $\mathcal{D}$ at all times. We set $\mathcal{M} := \mathcal{M}_{s'} \cap \mathcal{D}$ at step 7. Moreover, before we descend into a state s' at step 5 of the algorithm, we check to ensure that there exists some term in $\mathcal{D}$ that is a descendant of s'. Using these optimizations, we can identify failures early, and moreover avoid inspecting some positions that are not necessary to determine the candidate set.

A variant of path indexing is obtained by limiting the maximum lengths of path strings. Those path strings longer than this length are truncated. This variation has been proposed and studied by McCune [1992] in the OTTER system. By controlling the maximum length, we can control the size of the trie. The savings are particularly significant in McCune's version, since it stores the sets $\mathcal{C}_s$ at each node in the trie. As these sets are represented as lists, the storage required per node in the trie is substantial. Graf [1992] uses an alternative approach where the sets $\mathcal{C}$ are not stored, and this representation is less sensitive to this optimization. Other techniques for reducing the size, such as pruned and collapsed tries, have not been studied. As compared to length limiting the paths, the pruning and collapsing techniques have the advantage that no accuracy is lost by these techniques.

Graf [1992] proposes a variation of path indexing in which the candidate set elements are retrieved one-at-a-time. To accomplish this, we can explicitly construct a data structure (called the *query tree*) that captures the union and intersection operations performed by the retrieval algorithms presented above. In particular, as we traverse the trie, we construct the query tree that represents the set operations to be performed, rather than performing them directly. This query tree can then be evaluated to yield the candidate set elements one at a time. In particular, we can try to compute the first element in the candidate set, then the second element, and so on. One of the advantages of this approach is that it can deal with insertions to the index concurrent with the retrieval. Assume that the indexed terms are given integer identifiers in such a manner that terms created later on have a larger id than terms created earlier. We now evaluate the query tree to get the candidate terms in the increasing order of the id. This approach ensures that if new terms were to be added to the query tree in the middle of the retrieval process, these terms would get larger identifiers than the terms already existing in the index, and hence will be retrieved after all of the terms already in the index. This ability to process concurrent retrieval and insertion is particularly convenient in applications where the retrieved term may be processed in such a way that new terms may have to be inserted into the index.

Riazanov and Voronkov [2001*a*] discuss a modification of path indexing that also works for multiliteral clauses. Their modification also uses skip lists to store sets of literals or clauses at each node. There are no union operations since every node stores the list of all literals or clauses stored in the leaves descendent from this node. Intersection operations are optimized by changing the order of intersections and using fast traversal of skip lists.

5.4. Summary of advantages and disadvantages

Path indexing has been studied extensively by McCune [1992], Graf [1992], [Riazanov and Voronkov 2001*a*] in the context of automated theorem proving, and by Ramesh et al. [1990], Chen, Ramakrishnan and Ramesh [1992] in the context of logic programming. The slight variations in these implementations were outlined earlier. Specifically, McCune's [1992] implementation uses hashing instead of tries, and moreover, stores the sets $\mathcal{C}_s$. In contrast, Graf uses tries and also avoids storing the $\mathcal{C}_s$ sets. As such, the latter approach utilizes less memory than the former.

One of the main advantages of path indexing is that it is economical in terms of memory usage, more so than any other indexing technique discussed in this paper. The best performance in terms of memory usage is obtained when we use tries to represent the index, and store the candidate sets only at the leaves of the index, but not at the intermediate nodes. Another aspect of memory usage is that is can be further reduced by placing depth restrictions on indexing, or possibly by using techniques such as pruning and collapsing.

A second advantage is that path indexing involves no backtracking. Symbols in the query term are inspected at most once, thus leading to better retrieval time.

The insertion and deletion operations on the index are also very efficient, typically beating the times for insertion and deletion operations for other indexing techniques.

One of the main disadvantages is the cost of combining intermediate results. This leads to decreased retrieval performance. As compared to the other indexing techniques, path indexing can be useful for retrieving instances or implementing backward subsumption. The performance becomes worse for retrieving unifiable terms or generalizations.

6. Discrimination trees

6.1. Overview

In *discrimination tree* indexing, we generate a single p-string from each of the indexed terms. This p-string is obtained via a preorder traversal of the terms. We then build a trie consisting of these p-strings.

By exploiting the nature of preorder traversals, we can develop a more optimized representation for the p-strings. In particular, note that given that the function symbols have predefined arities, there is a unique correspondence between the string obtained by preorder traversal and the term, even when the position information is completely ignored. Thus, we can use a simplified representation where the position information is no longer used. Moreover, we can annotate the final states in the trie with the candidate set $\mathcal{M}$ corresponding to the state. There is no need for (potentially expensive) combination operations that were required in the case of path indexing.

We illustrate discrimination tree indexing with the following example.

$$\begin{array}{l}(1)\ f(g(a,*),c),\quad (2)\ f(g(*,b),*),\quad (3)\ f(g(a,b),a),\\ (4)\ f(g(*,c),b),\quad (5)\ f(*,*).\end{array}$$

The following p-strings are obtained from these terms. We have omitted the position information from the p-strings.

$$\begin{array}{ll} f.g.a.*.c & \{1\}\\ f.g.*.b.* & \{2\}\\ f.g.a.b.a & \{3\}\\ f.g.*.c.b & \{4\}\\ f.*.* & \{5\}\end{array}$$

The index for retrieval of generalizations of the query term $f(g(a,c),b)$ is shown in Figure 12. To understand the process of indexing, note that the string corresponding to the query term is $f.g.a.c.b$. We compare the symbols in this string successively with the symbols on the edges in the path from the root to state 5. At this point, we cannot take the left path, as the symbol b on this edge conflicts with the symbol c

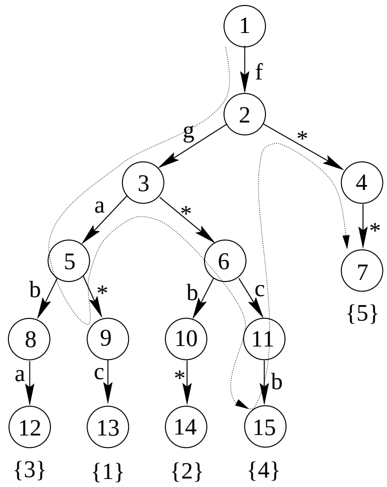


Figure 12: Example of discrimination tree indexing

in the query term. However, the symbol $*$ on the edge leading to state 9 is entirely plausible, since taking this edge corresponds finding a generalization (namely, a variable) of the subterm c of the query term. However, we cannot proceed further from state 9, so we have to backtrack to state 3. At this point, we can follow down the $*$ branch all the way down the final state 15, identifying candidate term 4. If we are interested in all generalizations, we have to backtrack further to state 2, and then finally follow down to state 7, identifying candidate term 5.

Finally, we note that in order to perform retrieval of unifiable terms and instances, we must efficiently deal with situations where the query term has a variable at a point where the indexed terms contain a function symbol. In such a case, we need a mechanism to efficiently skip the corresponding subterms in the indexed terms. We can make use of *jump lists* for this purpose.

Earliest known implementations of discrimination trees are due to Greenbaum [1986]. Subsequently, Christian [1989] developed the flatterm representation for use in discrimination trees, and this resulted in excellent speedups [Christian 1989, Christian 1993]. Discrimination trees have been further studied extensively by McCune [1992] and Graf [1996]. They are used extensively in the provers Otter, Waldmeister, and E.

6.2. Indexing algorithms

6.2.1. Index construction and maintenance

Construction of a discrimination tree is straightforward. We start with an empty tree, and successively insert each of the indexed terms into the tree. This is accom-

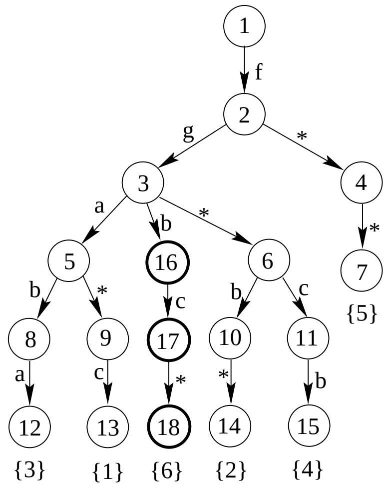


Figure 13: Insertion of term $f(g(b, c), *)$

plished by constructing the preorder string from the term to be inserted, and then inserting this string into the tree. Since the index is a trie, algorithms for inserting strings into the trie are well known and not discussed further here. Figure 13 illustrates the insertion operation on discrimination trees.

Deletion operation can also be performed readily, since it amounts to deleting the corresponding preorder string from a trie. Deletion operation is illustrated in Figure 14. Introduction of jump lists complicates the insertion and deletion algorithms. Effectiveness of jump lists in practice was not studied.

6.2.2. Term traversal operations

In this and following sections we introduce algorithms for several retrieval operations. These algorithms will use functions for term traversal introduced below.

For technical purposes we extend $\mathcal{P}(t)$ by a special object ε called the *end position* in t. The set $\mathcal{P}(t) \cup \{\varepsilon\}$ will be denoted by $\mathcal{P}^+(t)$. When it is necessary to tell the end position from the other positions, we call the positions in $\mathcal{P}(t)$ *proper positions*.

We denote by $<$ the lexicographic ordering on positions extended in the following way: $p < \varepsilon$ for any proper position p. To perform traversal of a term t we will need two operations on proper term positions: $next_t$ and $after_t$, which can be informally explained as follows. Represent the term t as a tree and imagine a term traversal in the left-to-right, depth-first direction. Suppose $t/p = s$. Then $t/next_t(p)$ is the subterm of t visited immediately after s, and $t/after_t(p)$ is the subterm visited immediately after traversal of all subterms of s. Formally, let $\Lambda = p_1 < \ldots < p_n < p_{n+1} = \varepsilon$ be all positions in t. Then $next_t(p_i) = p_{i+1}$ for all $i \leq n$. The definition of $after_t$ is as follows: $after_t(p_i) = p_j$, where j is the smallest number such that $j > i$ and for all $i < k < j$ the position p_i is a prefix of p_k.

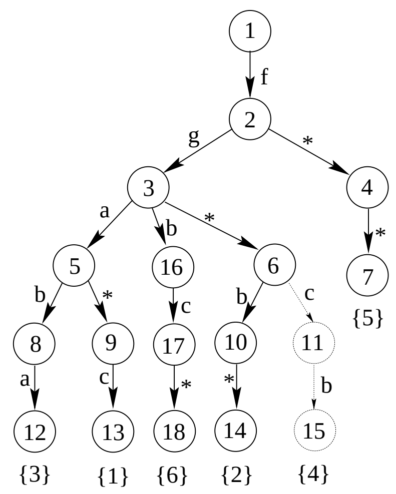


Figure 14: Deletion of term $f(g(*, c), b)$

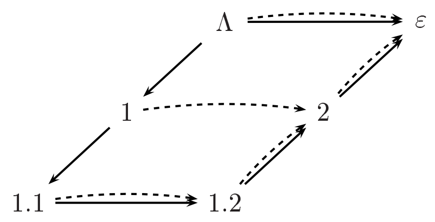


Figure 15: $next_t$ and $after_t$ on the positions in $t = f(f(a, a), a)$. Solid straight lines and dashed arcs depict $next_t$ and $after_t$ respectively.

Figure 15 illustrates the behavior of *next* and *after* on the positions in the term $f(f(a, a), a)$.

6.2.3. Retrieval of generalizations

An algorithm for retrieval of generalizations from a discrimination tree is shown in Figure 16.

Even though the automata for path indexing look quite different from those for discrimination tree indexing, the retrieval algorithms have much in common. In

function $retrieve$(index state V, term t, position p) returns set of terms $\mathcal{C}$
1. $\mathcal{C} := \phi$
2. if $\exists$ a transition from state V to another state V' labeled with $root(t/p)$ then
3.   if V' is a final state
4.   then $\mathcal{C} := \mathcal{C}_{V'}$
5.   else
6. $\mathcal{C} := retrieve(V', t, next_t(p))$
7.   endif
8. endif
9. if $\exists$ a transition from V to a state V_* labeled with $*$
10.   then $\mathcal{C} := \mathcal{C} \cup retrieve(V_*, t, after_t(p))$
11. return $\mathcal{C}$

Figure 16: Algorithm for retrieval of generalizations from a discrimination tree

particular, only steps 6 and 10 in the above algorithm are different from path indexing retrieval algorithm. Out of these two steps, the difference at step 6 arises due to (a) the fact that no intersection operations need to be performed in discrimination tree, and (b) because the next position to visit is implicit, and needs to be computed based on the current position being inspected and the query term t itself. The difference at step 10 arises because V_* would be a final state in a path index, whereas in a discrimination tree, it will have further descendants in general, and hence the trie needs to be traversed further. The second difference arises again because of implicit traversal, which in this case, requires us to skip all positions in u that are the children of p. The problem is caused by *embedded variables*, i.e., the variables at the position p in the query term such that some indexed terms have a nonvariable at the position p.

Note that instead of returning the entire set, it may be preferable to return the candidate terms one at a time. This can be accomplished by using a backtracking algorithm. In particular, instead of computing a union at step 10, we would set a choice point on the trail stack. Later on, in order to retrieve the next candidate term, we would backtrack to this choice point, and then explore the '$*$' transition.

6.2.4. Retrieval of unifiable terms

An algorithm for retrieval of unifiable terms from a discrimination tree is shown in Figure 17. In this figure we denote by $JumpList(V)$ the jump list for a state V.

Note again that the algorithm for retrieval of unifiable terms is also similar to the corresponding algorithm for path indexing. The differences arise mainly because of the reasons as before: the traversal order is implicit in discrimination trees, so the next position to visit has to be computed explicitly. Also, when a variable is inspected in the query term, we need to skip the corresponding portions of the indexed terms, which is accomplished using the jump lists.

Figure 18 illustrates this algorithm. We need to make use of the jump lists for efficiently skipping portions of the discrimination tree that correspond to a variable

```
function retrieve( index state V, term t, position p) returns set of terms C
if t/p = '*'
    then C = ⋃_{V' ∈ JumpList(V)} retrieve(V', t, next_t(p))
else
    C := φ
    if ∃ a transition from state V to a state V' labeled with root(u/p) then
        if V' is a final state
            then C := C_{V'}
        else
            C := retrieve(V', t, next_t(p))
        endif
    endif
    if ∃ a transition from V to a state V_* labeled with *
        then C := C ∪ retrieve(V_*, t, after_t(p))
endif
return C
```

Figure 17: Algorithm for retrieval of unifiable terms from a discrimination tree

in the query term. In the general case, there is a jump link from every node in the tree to all its descendent states that examine the position immediately after all of the positions within the current subterm. The storage for such links can be substantial, and would clutter the picture. So, we have shown jump lists only for those nodes where jump links go to a state different from the immediate child of a node.

6.3. Variations

Perfect discrimination trees were proposed by McCune [1992]. Such trees deal with nonlinearity, and are hence perfect filters. To deal with nonlinearity, these trees use named variables, as opposed to the anonymous variable '*' in standard discrimination trees. However, using different variables in different terms will adversely affect the ability to share prefixes of preorder strings in the tree, and hence lead to extensive backtracking. To avoid this, variables in indexed terms are *normalized* so that the same set of variables can be used across different indexed terms in a consistent manner. For instance, we could use x_i to denote the ith distinct variable in an indexed term. Then the term $f(y, g(y, z))$ will be represented as $f(x_1, g(x_1, x_2))$. Another possibility is to linearize the term, i.e., represent every occurrence of a variable as a distinct variable and also store in the index *equality constraints* indicating which variables in the indexed term are equal. For example, the term $f(y, g(y, z))$ will be represented as $f(x_1, g(x_2, x_3))$ plus the equality constraint $x_1 = x_2$. Equality constraints are used in [Rivero 2000, Riazanov and Voronkov 2000*b*].

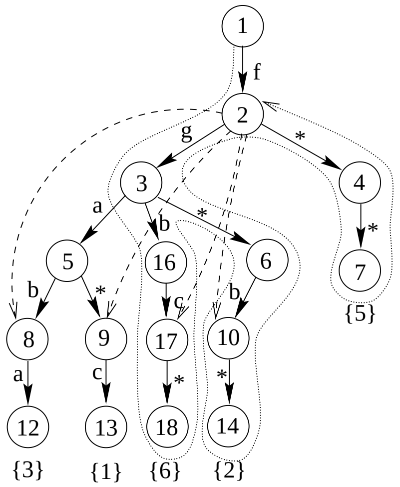


Figure 18: Retrieval of terms unifiable with $f(g(b,*),a)$

Depth-limiting [McCune 1992] is an approach to limiting the size of the discrimination tree, possibly at the expense of retrieval time.

Deterministic discrimination trees [Graf 1991] are a variation that avoids backtracking altogether. Thus, in a single scan of the "relevant portions" of the query term, we can determine all of the candidate terms. Deterministic trees have been proposed mainly in the context of retrieving generalizations. They avoid states that have transitions on variables and nonvariables, since such states necessitate backtracking. This is accomplished by selectively instantiating some of the variable positions in each indexed term t to obtain a set of instances $\mathcal{T}$ of the indexed term such that the set of ground instances of t is identical to the set of ground instances of all of the terms in $\mathcal{T}$. For instance, the set of terms

$$\{f(g(a,*),c), f(g(*,b),*), f(g(*,c),b)\}$$

would be expanded into the set

$$\{f(g(a,b),c), f(g(a,b),\neq), f(g(a,c),c), f(g(a,c),b), f(g(a,\neq),c), f(g(\neq,c),b)\},$$

where the symbol $\neq$ is a wildcard symbol that matches every function symbol except those occurring at that position in the original terms. Note that such expansion results in an explosion in the number of indexed terms — in fact, the blow up can be exponential. Deterministic automata are used in applications where the indexed terms change very infrequently, e.g., functional programming and term rewriting with (almost) fixed set of rules. We will consider a generalized version of deterministic automata in the next section.

6.4. Summary of advantages and disadvantages

Discrimination trees improve over path indexing in avoiding the expensive set intersection operations that are required to obtain candidate terms from candidate p-strings. One disadvantage is that they tend to use more storage, since we cannot share states for examining symbols at a position p from multiple indexed terms unless they have identical symbols in every position p' that precedes p in preorder traversal. Similar sharing in path indexing only requires that the terms are identical in positions that are ancestors of p. Space usage is exacerbated significantly if skip lists are maintained. Maintenance of skip links also makes insertion and deletion operations significantly more expensive. A second disadvantage is that backtracking is typically required in retrieval operations, necessitating reexamination of symbols. However, this overhead is typically small as compared to the cost of set intersection operations in path indexing.

Perfect discrimination trees improve on standard trees in their ability to incorporate tests for consistency of substitutions in the index. Moreover, the binding operations (i.e., operations for computing substitutions) can be shared across multiple indexed terms. However, since consistency checking operations can be very expensive (e.g., when the substitutions being compared are large), introduction of these operations into the index can degrade performance. It would be better to postpone these expensive operations so that they occur after the simpler operations of checking for the occurrence of a symbol at a position. Such reordering is possible with some of the techniques described later on in this paper.

Deterministic discrimination trees improve upon standard trees in that no backtracking is required for retrieval. The downside is that they can be very large – whereas the size of standard discrimination trees (measured as the number of nodes) is linear in the sum of sizes of indexed terms, the worst case size of deterministic trees can be exponential in the number of indexed terms. This also means that insertion and deletion operations in the index are expensive. Thus, deterministic trees are suitable primarily for applications where retrieval performance is important, and efficiency of maintenance operations is not a concern.

7. Adaptive automata

7.1. Overview

For constructing adaptive automata, we generate one p-string from each indexed term. The traversal order for generating the p-string is not fixed *a priori*, as in the case of discrimination trees. Instead, the traversal order is *adapted* to suit the set of indexed terms. The adaptation is designed to minimize the size of the automaton and the retrieval time.

Although adaptive automata can use backtracking, they have been studied primarily in the context of deterministic automata for retrieving generalizations. In particular, this means that no state in the automaton has a transition on a function

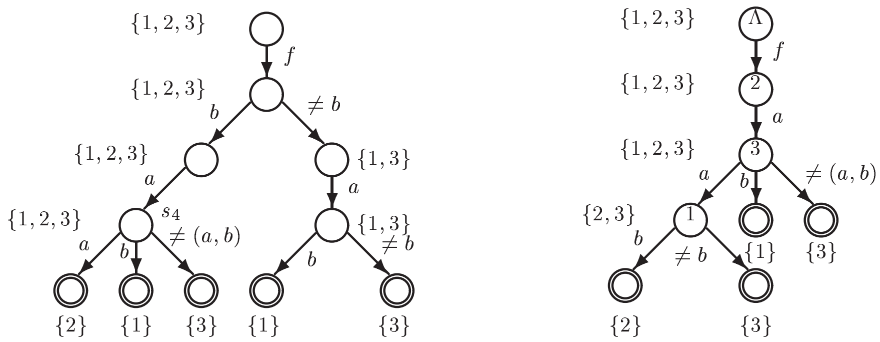


Figure 19: A left-to-right automaton and an adaptive automaton

symbol and a variable. The traversal order for generating the p-strings is designed in such a way as to avoid constructing such states. When such branches become unavoidable, the indexed term containing the variable is instantiated at this position with all possible symbols that can appear in this context. Index construction then proceeds with these instances in place of the term containing the variable.

In generating p-strings for adaptive automata, we ensure that all p-strings with a common prefix examine the same position after this prefix. When we construct a trie of such p-strings, we have a unique transition out of any automaton state that examines a position. Thus, we can optimize the representation by storing the next position to inspect as part of an automaton state, rather than creating a transition based on this position. A second optimization, which applies to all deterministic automata, is that the final states directly yield all possible candidate terms — there is no need to search any further.

In this section we assume that the set of indexed terms is prioritized, i.e., we have a function $priority(l)$, which allows us to compare priorities of indexed terms, i.e., checking if $priority(l) \geq priority(l')$ holds for given indexed terms l, l'. Let us give an example. Consider the set consisting of the following three terms:

$$(1)\ f(x,a,b), \quad (2)\ f(b,a,a), \quad (3)\ f(x,a,y),$$

where the terms with the smaller number have the largest priority. Figure 19 shows a left-to-right automaton and an adaptive automaton for this set of terms.

Adaptive traversals, as embodied in adaptive automata, possess the following advantages over fixed-order traversals:

- adaptive automata are typically smaller, e.g., 8 states vs. 11 states in the example. The reduction factor can even become exponential.
- retrieval requires lesser time, e.g., left-to-right automaton needs to inspect four positions to announce a match of the query term $f(c,a,b)$ against indexed term 1, whereas the adaptive automaton inspects only a proper subset of these posi-

tions. Examining unnecessary symbols is especially undesirable in the context of lazy functional languages.

The origins of adaptive indexing can be traced back to the development of complete normalization strategies for a subset of orthogonal rewrite systems [Dershowitz and Jouannaud 1990] called *strongly sequential systems* [Huet and Levy 1978, Huet and Levy 1991]. This work was extended for lazy functional languages in [Laville 1988, Laville 1987, Puel 1990, Maranget 1992, Kennaway 1990]. The technique was studied in the context of arbitrary terms in [Sekar, Ramesh and Ramakrishnan 1992].

7.2. Indexing algorithms

In this section, we develop algorithms for constructing adaptive automata for a prioritized set of indexed terms, and using them for retrieval of generalizations of a query term. The idea behind the construction of adaptive automata is the following. We "guess" the query term t position by position, and build an automaton for retrieval of generalizations of t using the information about currently known positions. The "so far guessed" part of t is a linear term that will be denoted by u. We have $u \leq t$; this implies that every indexed term l compatible with t (i.e., $l \leq t$) must also unify with u. Sometimes, we can find the match for t without complete inspection of t, but only using partial information about t available in u. This happens when we find out that some indexed term l is compatible with u but no indexed term l' of a higher priority is unifiable with u). This suggests the following definition.

7.1. Definition. A term $l \in \mathcal{L}$ *$\mathcal{L}$-matches* u if $l \leq u$, and no $l' \in \mathcal{L}$ with priority greater than that of l unifies with u. Given a term u, we define its *match set*, denoted by $\mathcal{L}_u$, as the set of terms in $\mathcal{L}$ unifiable with u.

Intuitively, $\mathcal{L}_u$ consists of all indexed terms that can potentially be generalizations of t. We will use the wildcard $\neq$ in the term u in the following way: $\neq$ unifies with any variable, but does not unify with a nonvariable symbol.

7.2.1. Index construction

The algorithm *Build* for constructing an adaptive automaton is shown in Figure 20. A state V of the automaton remembers the prefix u of a query term that would have been inspected while reaching that state from the start state. Suppose that p is the next position inspected from V. Then there are transitions from V on each distinct symbol c that appears at p for any $l \in \mathcal{L}_u$. There will also be a transition from V on $\neq$ which will be taken on inspecting a symbol different from those on the other edges leaving V.

The symbol $\neq$ appearing at a position p denotes the inspection of a symbol in the input that does not occur at p in any indexed term in $\mathcal{L}_u$. This implies that

Procedure *Build*(V, u)

1. let $\mathcal{M}$ denote the set of all indexed terms that match u.
2. if $\mathcal{M} = \{l\}$ and $\forall\, l' \in \mathcal{L}_u$ $\mathit{priority}(l) \geq \mathit{priority}(l')$ then
3.   mark V with $\{l\}$ and terminate
4. else
5. $p = \mathit{select}(u)$; /* *select* is a function to choose the next position to inspect */
6. $\mathit{pos}[V] = p$; /* Next position to inspect is recorded in the *pos* field */
7.   for each nonvariable symbol c for which $\exists l \in \mathcal{L}_u$ with $\mathit{root}(l/p) = c$ do
8.   create a new node V_c and an edge from V to V_c labeled c;
9. $\mathit{Build}(V_c, u[c(y_1, \ldots, y_n)]_p)$ /* $y_1, \ldots, y_n$ are new variables, n is the arity of c */
10.   if $\exists l \in \mathcal{L}_u$ with a variable at p or above p then
11.   create a new node $V_{\neq}$ and an edge from V to $V_{\neq}$ labeled $\neq$;
12. $\mathit{Build}(V_{\neq}, u[p \leftarrow \neq])$

Figure 20: Construction of adaptive automata

if a prefix u has $\neq$ at a position p then every indexed term that could potentially match an instance of u must have a variable at or above p.

Procedure *Build* is recursive, and the automaton is constructed by invoking $\mathit{Build}(s_0, x)$ where s_0 is the start state of the automaton. *Build* takes two parameters: V, a state of the automaton and u, the prefix examined in reaching V. The invocation $\mathit{Build}(V, u)$ constructs the subautomaton rooted at V.

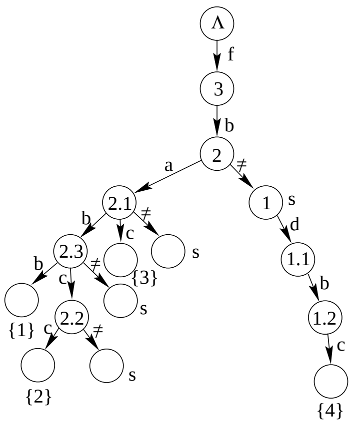
At line 2, the termination condition is checked. By the definition of indexed term match, we need to rule out possible matches with higher priority indexed terms before declaring a match for a lower priority indexed term. Since the match set $\mathcal{L}_u$ contains all indexed terms that could possibly match the prefix u, we simply need to check that each indexed term in the match set is either already in l or has a lower priority than l.

If the termination conditions are not satisfied then the automaton construction is continued at lines 5 through 12. At line 5, the next position p to be inspected is selected and this information is recorded in the current state in line 6. Lines 7, 8 and 9 create transitions based on each symbol that could appear at p for any indexed term in $\mathcal{L}_u$. In line 9, *Build* is recursively invoked with the prefix extended to include the symbols seen on the transitions created at line 8. If there is an indexed term in $\mathcal{L}_u$ with a variable at or above p then a transition on $\neq$ is created at line 11 and *Build* is recursively invoked at line 12. The recursive calls initiated at lines 9 and 12 together will complete the construction of the subautomaton rooted at state V. An example adaptive automaton built using this algorithm is shown in Figure 21.

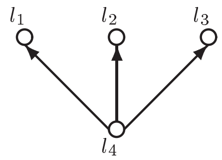
7.2.2. Retrieval of generalizations

An algorithm for retrieval of generalizations is shown in Figure 22.

As compared to discrimination trees, the retrieval algorithm is further simplified: there is no backtracking involved, and the final state reached directly yields all of



$l_1 : f(*, a(b, *, b), b)$
$l_2 : f(*, a(b, c, c), b)$
$l_3 : f(*, a(c, *, *), b)$
$l_4 : f(d(b, c), *, b)$



1. States are labeled by the next position to be inspected upon reaching that state.
2. All of the states marked with an s actually denote a subautomaton that is identical to the automaton rooted at the state labeled with 1.

Figure 21: Example of an adaptive indexing automaton

function $retrieve(IndexState\ \ V, Term\ \ t)$ returns $SetOfTerms$
if $\exists$ a transition from V to another state V'
    labeled with $root(t/pos[V])$ or else with $\neq$ then
  if V' is a final state labeled with $\{l\}$ then
    return $\{l\}$
  else
    return $retrieve(V', t)$
  endif
else
  return $\emptyset$
endif

Figure 22: Retrieval of generalizations from adaptive automata

the candidate terms. In addition to specifying all of the candidate terms, the final state can also store the substitution for each of the candidate terms, i.e., there would be no need to explicitly compute the substitution at retrieval time.

7.2.3. Time and space complexity

The primary objective of a selection function is to reduce the automaton size and/or the matching time. Therefore it is important to know how we measure these quantities. A natural measure of the size of an automaton is the number of states in it. However, this measure has the drawback that minimization of total number of states is NP-complete, even for the simple case of indexed terms with no variables [Comer and Sethi 1976]. This makes it impossible to develop efficient algorithms that build an automaton of smallest size, unless P = NP. Even so, we would still like to show that certain algorithms are always better than others for reducing the size. The usual way to do this is to choose an alternative measure of size that is closely related to the original size measure, yet does not have the drawback of NP-completeness of its minimization. A natural choice in this case is the breadth of the automaton, which is closely related to the total number of states.

As for matching time, it is easy to define the time to match a given term using a given automaton: it is simply the length of the path in the automaton from the root to the final state that accepts the given term. However, what we would like is a time measure that does not refer to input terms. We could associate an average matching time with an automaton, but this would require information that is not easily obtained: the relative frequencies with which each of the paths in the automaton are taken[3]. Therefore, instead of defining a time measure that totally orders the automata for a specific distribution of input terms, we use the following measure that partially orders them *independent* of the distribution. Let $MT(s, A)$ denote the length of the path in (automaton) A from the start state to the accepting state of the *ground* term s. If s is not accepted by A then $MT(s, A)$ is undefined.

We now examine upper and lower bounds on the space and matching time complexity of adaptive tree automata for several classes of terms. Since the traversal order itself is a parameter here, we first need to clarify what we mean by upper and lower bounds. By an upper bound, we refer to an upper bound obtained by using the best possible traversal for a set of indexed terms, i.e., a traversal that minimizes space (or time, as the case may be). The rationale for this definition is that for every set of indexed terms, there exist traversal orders that can result in the worst possible time or space complexity. Clearly, it is not interesting to talk about the upper bound on size of the automaton obtained using such a (deliberately chosen) nonoptimal traversal order. Our lower bounds refer to the lower bounds obtained for any possible traversal order.

7.2.4. Greedy strategies for minimizing space

Representative Sets. Consider the indexed terms in Figure 21 and the prefix $u = f(*, a(b, *, b), *)$. Although $\mathcal{L}_u = \{l_1, l_4\}$, observe that a match for l_4 can be declared only if the 3^{rd} argument of f is b. In such a case we declare a match for the higher priority indexed term l_1.

Inspecting any position only on behalf of a indexed term such as l_4 is wasteful, e.g., inspection of position 1 for u is useless since it is irrelevant for declaring a match for l_1. We can avoid inspecting such positions by considering the representative set instead of a match set for a prefix u. A representative set is defined formally as follows:

7.2. DEFINITION. A *representative set* $\overline{\mathcal{L}_u}$ of a prefix u with respect to a set of indexed terms $\mathcal{L}$ is a minimal subset $\mathcal{S}$ such that the following condition holds for every l in $\mathcal{L}$:

[3]It is possible to assume that all terms over Σ are equally likely and derive a matching time on this basis, but such assumptions are seldom justified or useful in practice.

Class of Terms	Lower bound on space	Upper bound on space	Lower bound on time	Upper bound on time
Unambiguous, no priority	$\Omega(2^{\sqrt{\alpha}})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(\alpha)$	S
Unambiguous, with priority	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(S)$	S
Ambiguous	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(S)$	S

Notation

l_i : i^{th} indexed term

n : Number of Indexed Terms

S : Total number of nonvariable symbols in indexed terms

α : Average number of nonvariable symbols in indexed terms

Figure 23: Space and matching time complexity of adaptive automata.

$$\forall t \geq u \ \ (l \leq t) \Rightarrow \ \exists l' \in \mathcal{S} \ \ [(l' \leq t) \wedge (\mathit{priority}(l') \geq \mathit{priority}(l))]. \tag{7.1}$$

All these strategies select the next position based on *local information* such as the prefix and the representative set associated with v or its children. Let p denote the next position to be selected.

1. Select a p such that the number of distinct nonvariables at p, taken over all indexed terms in $\overline{\mathcal{L}_u}$ is *minimized*. This strategy attempts to minimize the size by local minimization of breadth of the automata. It does not attempt to reduce matching time.
2. Select a p such that the number of distinct nonvariables at p, taken over all indexed terms in $\overline{\mathcal{L}_u}$ is *maximized*. The rationale here is that by maximizing the breadth, a greater degree of discrimination is achieved. If we can quickly distinguish among the indexed terms, then the (potentially) exponential blow-up can be contained. Furthermore, once we distinguish one indexed term from the others, we no longer inspect unnecessary symbols and so matching time can also be improved.
3. Select a p such that the number of indexed terms having nonvariables at p is maximized. The motivation for this strategy is that only indexed terms with variables at p are duplicated in the representative sets of the descendants of the current state v of the automaton. By minimizing this number of indexed terms that are duplicated, we can contain the blow-up. Furthermore, this choice minimizes the probability of inspecting an unnecessary position: it is a necessary position for the most number of indexed terms.

4. Let $\overline{\mathcal{L}_1}, \ldots, \overline{\mathcal{L}_r}$ be the representative sets of the children of v. Select a p such that $\Sigma_{i=1}^{r}|\overline{\mathcal{L}_i}|$ is minimized. Note that the main reason for exponential blow-up is that many indexed terms get duplicated among the representative sets of the children of v. This strategy locally minimizes such duplication (since $\Sigma_{i=1}^{r}|\overline{\mathcal{L}_i}|$ is given by the size of $\overline{\mathcal{L}_u}$ plus the number of indexed terms that are duplicated among the representative sets of the children states.) For improving time, this strategy again locally minimizes the number of indexed terms for which an unnecessary symbol is examined at each of the children of v.

All of the above greedy strategies suffer from the drawback that:

7.3. THEOREM. *For each of the above strategies there exist indexed term sets for which automata of smaller size can be obtained by making a choice different from that given by the strategy.*

The contrived nature of the example shows that although it is possible for these strategies to fail, such failures may be atypical. Even when they fail, as in the above example, they still appear to be significantly better than fixed order traversals. For instance, right-to-left traversal constructs an automaton with breadth 30 and number of states 60. Left-to-right traversal reduces these figures to 24 and 58, which is still more than that obtained by using any of the strategies 2,3 and 4.

7.2.5. Reducing matching time by selecting index positions

We now propose another important local strategy that does not suffer from the drawbacks of the greedy strategies discussed in the previous section. The key idea is to inspect the index positions in u whenever they exist. We show that this strategy yields automata of smaller (or same) size and superior (or same) matching time than that obtainable by any other choice.

We call an *index position* any position p in the fringe of u such that every instance of u for which there is a match in $\mathcal{L}$, u/p is a nonvariable. A set of terms is called *constructor-based* if the outermost symbol in every term is different from all of the nonoutermost symbols in all other terms. It is strongly sequential if every prefix of every indexed term has an index position.

7.4. THEOREM. *Adaptive automata are space and time optimal for strongly sequential constructor systems.*

This result is very important in the context of complete normalization strategies for constructor-based orthogonal systems. Such systems form the basis of lazy functional programming languages such as Haskell and Hope. In lazy functional languages, evaluation (of input terms) is closely coupled with matching. Specifically, a subterm of the input term is evaluated only when its root symbol needs to be inspected by the matcher. If there are subterms whose evaluation does not terminate, then an evaluator that uses an algorithm that identifies matches without inspection of such subterms can terminate, whereas use of algorithms that do inspect such subterms will lead to nontermination. Since the set of positions inspected to identify a match is dependent on the traversal order used, the termination properties also depend upon the traversal order. In order to make sure that the program terminates on input terms of interest to the programmer, the programmer may have to reason about the traversal order used. In particular, the programmer can code his/her program in such a way that (for terms of interest to him/her) the matcher will inspect only those subterms whose evaluation will terminate. This implies that the programmer must be made aware of the traversal order used *even before the program is written* — thereby ruling out synthesis of *arbitrary* traversal orders at compile time. Given this constraint on preserving termination properties, a natural question is whether the traversal order can be "internally changed" by the compiler in a manner that is transparent to the programmer. Thus, given a traversal order T that is assumed by a programmer, can we make use of another traversal order S internally such that S is typically better than T, and is formally guaranteed to be no worse than T. We define such a traversal order below, based on the notion of index positions.

Given a traversal order T, denote by $S(T)$ the traversal order such that, for a prefix u, $S(T)$ selects the next position p in fringe of u such that:

- p is an index position, if u has index positions;
- p that is chosen by T, otherwise.

7.5. Theorem. *(i) $S(T)$ is no worse than T in terms of space consumption as well as retrieval time. (ii) In lazy functional programs, every program that terminates when matching using T is used will also terminate with $S(T)$.*

7.2.6. Minimizing space usage using DAG-automata

One of the main reason for the exponential space requirement is the use of tree structure in representing the automaton. Lack of sharing in trees results in duplication of functionally identical subautomata leading to wastage of space. A natural solution to this problem is to implement sharing with the help of dag structure (instead of tree).

An obvious way to achieve sharing is to use standard FSA minimization techniques. A method based on this approach first constructs the automaton (using algorithm *Build*) and then converts it into a (optimal) dag. However, the size of the tree automaton can be exponentially larger than that of the dag automaton. Therefore use of FSA minimization technique is bound to be very inefficient. To overcome this problem we must construct the dag automaton without generating its tree structure first. This means we must identify equivalence of two states *without even generating* the subautomata rooted at these states.

Central to our construction (of dag automaton) is a technique that detects equivalent states based on the representative sets. Consider two prefixes u and u' that have the same representative set $\overline{\mathcal{L}_u}$. Suppose that u and u differ only in those positions where every indexed term in $\overline{\mathcal{L}_u}$ has a variable. Since such positions are irrelevant for determining a match, these two prefixes are equivalent. On the other hand, it can also be shown that if they have different representative sets or differ in any other position then they are not equivalent. Based on this observation, we define the relevant prefix of u as follows. Let $p_1, p_2, \ldots, p_k$ denote (all of the) positions in u such that for each p_i there is at least one indexed term in $\overline{\mathcal{L}_u}$ that has a variable at p_i and all other indexed terms in $\overline{\mathcal{L}_u}$ have a variable either at p_i or above it. The relevant prefix of u is then

$$u[\neq]_{p_1}[\neq]_{p_2} \cdots [\neq]_{p_k}.$$

7.6. Theorem. *The automaton obtained by merging states with identical relevant prefixes is optimal.*

Merging equivalent states as described above can substantially reduce the space required by the automata, e.g., the tree automaton in Figure 21 has 25 states which can now be reduced to 16 by sharing.

We can show that the upper bound on size of dag automata is $O(2^n S)$ which is much smaller than the corresponding bound $O(\prod_{i=1}^{n} |l_i|)$ for tree automata. To prove this result, consider a dag automaton based on left-to-right traversal of indexed terms. We can also establish a lower bound of $O(2^\alpha)$ for ambiguous indexed terms. For unambiguous indexed terms, it is not clear whether the lower bound on size is exponential.

7.2.7. Computational complexity for building adaptive automata

Some of the central problems in computing adaptive automata are the computation of index positions and computation of representative sets. Both problems can be solved in quadratic time in the worst case for untyped languages. However, in the presence of types:

- Computing index positions is coNP-complete for typed terms, but there exists a polynomial time algorithm for untyped terms.
- Computing a representative set is NP-complete for typed terms, but takes polynomial time for untyped terms.

7.3. Summary of advantages and disadvantages

The primary advantage of adaptive automata are that they can be more compact and faster than discrimination trees. Like deterministic discrimination trees, they require no backtracking, but in general, this is achieved with a much smaller automaton. A disadvantage is that index maintenance operations may be more expensive. However, this is hard to assess, since the cost of these operations is closely related to the size of the index. Note that in order to retain the benefits of adaptivity in the presence of maintenance operations, the traversal order embodied in the automaton needs to be modified after insertion or deletion of terms. This may be done using the "interchange" operations outlined earlier, but these steps are in general expensive. However, they lead to reductions in space usage, which will in turn reduce the cost of future maintenance operations.

The benefits of adaptive automata outweigh the costs in applications where retrieval cost is to be minimized, while the cost of index construction or maintenance operations is not a concern. This is particularly true in declarative language implementations. Adaptive indexing technique provides the basis for complete evaluation algorithms for (lazy) functional languages.

While the size of adaptive automaton can be large in general, the algorithm for sharing equivalent states reduces this space requirement significantly. The same algorithm is applicable for deterministic discrimination trees as well. The space savings produced by this algorithm are more significant in the case of deterministic discrimination trees.

Backtracking adaptive automata provide an alternative approach that does not suffer from the space blowups associated with deterministic automata. The insertion and deletion operations are also rendered faster. Backtracking adaptive indexing still provides advantages over discrimination trees in terms of space usage as well as size.

8. Automata-driven indexing

8.1. Overview

Automata-driven indexing is an indexing technique that is based on string-matching. When a query term is checked for compatibility with a set of indexed terms, the substitution operations may have to be repeated for each candidate term, as the substitutions computed for different indexed terms may be different. For this reason, this technique focuses exclusively on sharing of the comparison operations that involve nonvariable symbols in the indexed and query terms. Specifically, the index in this approach is built from *preorder strings*, obtained from each indexed term as follows:

- traverse the term in preorder;
- break the p-string thus obtained at variable positions to get several strings, possibly as many as the number of variables in the term plus one.

For instance, consider the query term and indexed term shown in Figure 24, which will be used as the running example to illustrate the technique in the rest of this section. We have deliberately used an example with just a single indexed term in order to simplify the illustrations. The indexed term $f(g(a, a, h(y)), h(a))$ shown in the figure gives rise to two preorder strings: $\langle\Lambda, f\rangle\langle 1, g\rangle\langle 1.1, a\rangle\langle 1.2, a\rangle\langle 1.3, h\rangle$ and $\langle 2, h\rangle\langle 2.1, a\rangle$. The term $f(a, x, y)$ (not shown in the figure) generates just a single preorder string $\langle\Lambda, f\rangle\langle 1, a\rangle$.

In automata-driven indexing, a string-matching automaton is constructed from the preorder strings obtained from the indexed terms. The preorder strings from the query term are run through this automaton. The automaton states reached in this process capture all of the information relating to the nonvariable symbols in the query term. This information will be sufficient to determine the candidate terms from the indexed set, i.e., there will be no need to examine the query term

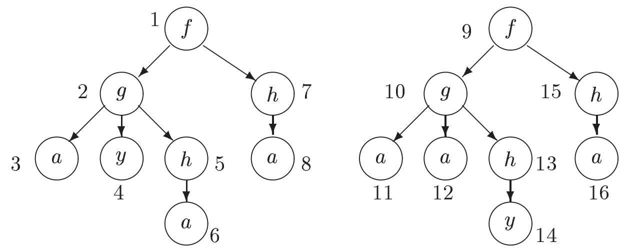


Figure 24: A query term s and an indexed term t.

symbols again. In principle, this technique can be used to retrieve unifiable terms, generalizations as well as instances. Another important feature of this technique is that it can be easily applied to retrieve indexed terms that are instances of (or are unifiable with) all subterms of the query term, rather than being limited to root matches only [Ramesh, Ramakrishnan and Sekar 1994]. We will, however, limit ourselves to the problem of unifiability of indexed terms with a query term.

Note that the preorder strings obtained in the manner described above do not fit the definition of p-strings, since some of the preorder strings can contain descendant positions without containing the ancestor positions. We can rework the definition of p-strings to accommodate this, but it is in fact simpler to define the notion of unifiability with respect to preorder strings.

8.1. Definition *(preorder string unifiability).* A preorder string $S \equiv \langle p_1, s_1\rangle\langle p_2, s_2\rangle \cdots \langle p_n, s_n\rangle$ is said to be *unifiable* with another preorder string $S' \equiv \langle p'_1, s'_1\rangle\langle p'_2, s'_2\rangle \cdots \langle p'_m, s'_m\rangle$ if and only if $\forall 1 \leq i \leq n\ \forall 1 \leq j \leq m\ \ p_i = p'_j \Rightarrow s_i = s'_j$.

The following preorder strings result from the terms shown in Figure 24:
S_s^1: $\langle \Lambda, f\rangle\langle 1, g\rangle\langle 1.1, a\rangle$
S_s^2: $\langle 1.2, h\rangle\langle 1.2.1, a\rangle\langle 2, h\rangle\langle 2.1, a\rangle$
S_t^1: $\langle \Lambda, f\rangle\langle 1, g\rangle\langle 1.1, a\rangle\langle 1.2, a\rangle\langle 1.3, h\rangle$
S_t^2: $\langle 2, h\rangle\langle 2.1, a\rangle$.

The above definition of preorder string unifiability requires that two preorder strings agree at every position that is common among them. Due to the nature of preorder strings, note that the subset of common positions must occur together. Therefore preorder string compatibility can be reduced to questions involving string matching. Specifically, four scenarios arise when checking the unifiability of two preorder strings, as shown in Figure 25. In this figure, the preorder string S_t is from an indexed term t, whereas S_s is a preorder string from the query term s. These two strings are shown so that they are aligned with each other at the common positions. p denotes the first common position among S_t and S_s, while l_t and l_s represent their respective lengths from this common position p. From the figure, it is easy to see

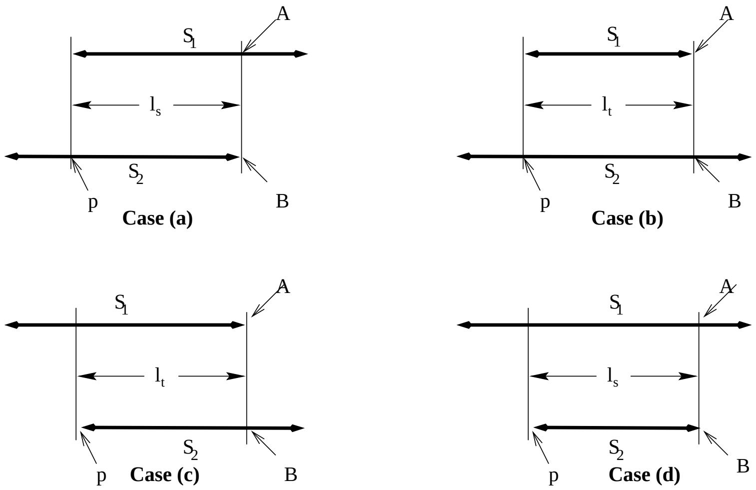


Figure 25: Scenarios for preorder string compatibility

that the string matching questions that need to be answered to determine preorder string unifiability are:

Q1: (Cases (a) and (b)) Does a prefix of S_t of length $\min(l_t, l_s)$ occur in S_s at p? In the example of Figure 24, case (b) arises between S_t^1 and S_s^1, with $p = \Lambda$, $l_s = 3$ and $l_t = 5$.

Q2: (Cases (c) and (d)) Does a prefix of S_s of length $\min(l_t, l_s)$ occur in S_t at p? In the example, case (c) arises between S_s^2 and S_t^2, with $l_s = l_t = 2$.

The index is built in such a fashion that all the information required to answer these questions can be obtained in a single scan through the query term. After this, each of these string matching questions will be answered in $O(1)$ time.

The origins of automata-driven indexing arose in the context of tree pattern matching [Ramesh and Ramakrishnan 1992]. It was then extended to deal with indexing of Prolog clauses in [Ramesh et al. 1990]. An implementation of this technique was developed [Chen, Ramakrishnan and Ramesh 1994] and integrated into the ALS Prolog system. Finally, it has been extended to the problem of retrieving clauses that are unifiable with a given query term or any of its subterms [Ramesh et al. 1994].

8.2. Indexing algorithms

8.2.1. Index construction

Index construction proceeds by first constructing all of the preorder strings corresponding to the indexed terms. Since the position information is redundant in preorder strings, this information is dropped. For the indexed term

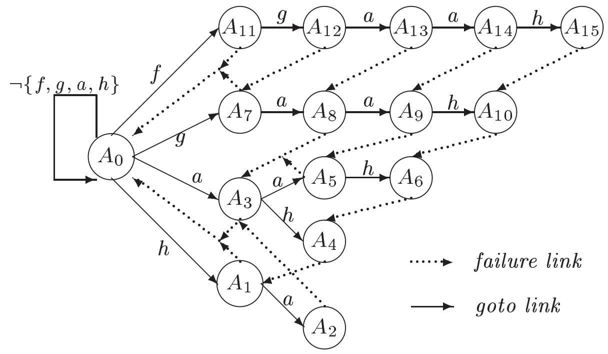


Figure 26: Automaton for suffixes of strings generated from indexed term.

$f(g(a, a, h(y)), h(a))$ shown in Figure 24, this leads to the strings $fgaah$ and ha.

We then build an Aho-Corasick automaton to recognize these strings, as well the substrings of these strings. This automaton serves as the index that supports retrieval of terms unifiable with a set of indexed terms. (For the rest of this section, we use the "automaton" and "index" interchangeably.) Figure 26 shows the automaton obtained for the preorder strings for the above indexed term. This automaton has two types of links: *goto* and $failure$. The *goto* links are forward links that are taken whenever we see a symbol in the input term that matches the symbol associated with the link. The $failure$ link is taken when the input symbol does not match the symbol associated with any of the forward links.

From the Aho-Corasick automaton, we can obtain a *goto tree* by deleting all the goto links. Similarly, we obtain a *fail tree* by deleting all the forward links and *reversing* the fail links. The fail tree for the automaton in Figure 26 is given in Figure 27(a). We say that the state A in the automaton represents string S if the (unique) path in the goto tree between the root (i.e., start state) and A spells S. For example, state A_9 in the automaton represents gaa.

The following properties of the automaton are essential for answering Q1 and Q2.

1. Every substring of every preorder string from every indexed term is represented by a unique state in the automaton. This implies: (a) each prefix of an indexed string is represented, and (b) every prefix of a query string that occurs in any indexed string is represented.
2. While scanning the string $a_1a_2 \ldots a_n$ if the automaton reaches a state A that represents a string S on reading a_j then S is the longest suffix of $a_1a_2 \ldots a_j$ among the strings represented by the automaton states.
3. If S_1 and S_2 are the strings represented by states A and B respectively then S_1 is a suffix of S_2 iff A is an ancestor of B in the fail tree.

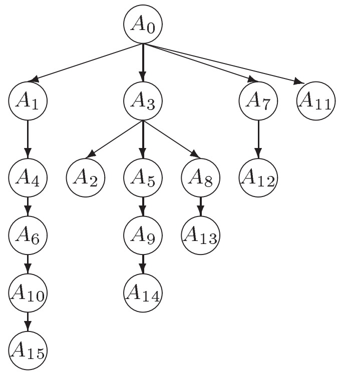


Figure 27: Fail tree corresponding to automaton of Figure 26

8.2.2. Retrieval of unifiable terms

The selection begins by scanning the preorder strings from the query term using the Aho-Corasick automaton and recording with each symbol the state reached upon reading it. Figure 28 shows the states of the index reached for our running example. The figure also shows the states reached on scanning preorder strings from the indexed term, this information being known at automaton construction time.

We now recall the string-matching questions that arise and describe how we can answer them (see Figure 25):

Q1: Does a prefix of S_t of length $l \equiv min(l_t, l_s)$ occur in S_s at p? We (i) obtain the state A representing the prefix of S_t of length l in the automaton, (ii) obtain the state B stored at $(p+l)$th position in S_q, and (iii) verify if A is an ancestor of B in the fail tree.

Q2: Does a prefix of S_s of length $l \equiv min(l_t, l_s)$ occur in S_t at p? We (i) obtain the state A reached on inspecting a prefix of S_t that extends to a length of l after the position p; (ii) obtain the state B representing the state reached on examining the prefix of S_s of length l; (iii) ensure that B actually represents the prefix of S_s of length l, i.e., ensure that $depth(B) = l$ in the goto tree[4]; (iv) verify whether B is an ancestor of A.

The detailed algorithm for retrieving unifiable terms is shown below in Figure 29. Procedure *Index* uses two arrays T and S that contain information about the symbols inspected in a preorder traversal of t and s respectively. These arrays are thus indexed by preorder numbers of nodes in the terms t and s. Each record in T has four fields: *label*, *varposn*, *subtree* and *state*. The *label* field is used to

[4]This step is necessary since it is possible that failure transitions may have been taken while examining the prefix of S_s. In such a case, B would represent a proper substring of S_s and not a prefix. The depth check ensures that no failure transitions were taken. Note that this check is not required for preorder strings from the indexed term, as such strings are known to be represented in the index.

Indexed-term info (computed at index construction time)	states	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	?	A_1	A_2
	preorder	f	g	a	a	h	Y	h	a
Query-term info (computed at retrieval time)	preorder	f	g	a	X	h	a	h	a
	states	A_{11}	A_{12}	A_{13}	?	A_1	?	A_1	A_2

Figure 28: Illustration of states reached when the strings from the indexed term and query term are scanned using the automaton of Figure 26

specify the functor/variable symbol at the ith position in a preorder traversal of the indexed term t. The *varposn* field at $T[i]$ is set to the preorder number of the nearest variable node that appears after i in preorder in t. The *subtree* field of $T[i]$ is set to the preorder number of the last node in the subtree rooted at node i. The *state* field specifies the state of the automaton reached on reading the symbol at i while scanning t. The structure of array S is identical to T. In addition the algorithm uses variables n_s, n_t, l_s, l_t and v_s. The variables n_s and n_t correspond to preorder numbers of nodes in s and t respectively up to which the retrieval procedure has proceeded without failure. The variables l_s and l_t store the lengths of remaining portions of query and indexed strings. v_s is set to true if the immediately preceding substitution was made to a variable in s. pf and nd are functions that return the preorder number and the number of descendants of a state in the fail tree respectively whereas function *depth* returns the depth of a state in the goto tree.

At run time the query term s is scanned prior to selection of any rule and all the fields in each record of S are filled. Note that T is filled at index construction time. Now procedure *Index* is then invoked to select t.

We illustrate the above method using Figure 28. After initialization at step 1, we proceed to step 10 where l is set to the minimum of the lengths of indexed term and query term strings. In this case, the query string is smaller ($l = 3$). We proceed through to step 15, where we verify that the state A_{13} representing the prefix fga is (trivially) an ancestor of the state A_{13} stored with the 3$^{\text{rd}}$ symbol in the query string. The algorithm loops back to step 3, where we skip the variable in the query term. Also, v_s is set at step 4. The algorithm proceeds to step 10. Here, l is set to 1, which corresponds to the remaining length of the indexed string. Since v_s is set, we proceed through steps 17 through 19, where we check whether the second query-term string prefix h occurs at the fifth position in the first string from the indexed term. Clearly, A_1 represents the prefix h (see Figure 43(c)). Furthermore, the state representing the first string from indexed term, namely A_{15}, is a descendant of A_1 in the fail tree. Hence the second string-matching step also succeeds. In the next string matching step, we look for the occurrence of the second string from indexed term starting at the 3$^{\text{rd}}$ position in the second string from the query term. Again we note that the state stored with the last symbol in the second string from the query term is the same as that representing the second string from the indexed term. Therefore this string-matching step also succeeds and the indexed term is selected to be included as a candidate term.

Procedure *Index*

```
n_s := n_t := 1; fail := false;
while (¬fail)∧ (s and t are not completely scanned) do
    if s[n_s].label is a variable then
        v_s := true;
        n_s := n_s + 1; n_t := T[n_t].subtree + 1;
    elsif T[n_t].label is a variable then
        v_s := false;
        n_t := n_t + 1; n_s := S[n_s].subtree + 1;
    else
        l := min(S[n_s].varposn − n_s + 1, T[n_t].varposn − n_t + 1);
        n_s := n_s + l; n_t := n_t + l
        pre_s := pf(S[n_s − 1].state); pre_t := pf(T[n_t − 1].state);
        if ¬v_s then /* instance of Q1 */
            nd_t := nd(T[n_t − 1].state);
            fail := ¬(pre_t ≤ pre_s ≤ pre_t + nd_t);
        else /* instance of Q2 */
            nd_s := nd(S[n_s − 1].state);
            d_s := depth(S[n_s − 1].state);
            fail := ¬(pre_s ≤ pre_t ≤ pre_s + nd_s) ∨ ¬(l_s = d_s)
        endif
    endif;
end
```

Figure 29: Procedure *Index* for retrieval of unifiable terms

When multiple indexed terms are present, we sequence through the indexed terms one-by-one, asking this series of questions on behalf of each indexed term. Note that although the questions are asked on a per-indexed-term basis, the string matching operations themselves are performed just once, and the symbols from the query term are also inspected at most once. Although this operation of sequencing through the indexed terms may appear very inefficient, it is in general unavoidable in techniques that generate multiple strings from each indexed term, such as path indexing and automata-driven indexing. Moreover, the impact of such sequencing can be minimized in practice using “coarse filtering” techniques (such as those described below) that quickly filter out indexed terms that are candidates.

8.2.3. Implementation issues

A straightforward way to perform rule selection is to invoke function *retrieve* once for every rule. However such a method regards every indexed term as a likely candidate, and will hence waste time on many indexed terms that can be readily ruled out. To do this, the above algorithm can be modified as follows. Specifically, we can construct a coarsely filtered set of indexed terms such that the first string of every one of these indexed terms is either a prefix of the first string from the query term or vice versa. This is done by taking all of the first preorder strings from all of the indexed terms and constructing an automaton to recognize these strings[5]. Let A be any state of the automaton, and let S_A be the string represented by A. Then this state is annotated with the set $\mathcal{M}$ of indexed terms such that for every $l \in \mathcal{M}$, the first string of l is a prefix of S_A or vice-versa. For retrieval, we traverse the coarse filtering automaton with the query term, stopping either when we encounter a final state of the automaton or when we reach the end of the first string in the query term. The set of indexed terms associated with the automaton state at this point will be taken as the coarse-filtered set of indexed terms. We can now restrict our attention to this subset of indexed terms, sequencing through them to answer string-matching questions for the subsequent strings in these indexed terms.

Several other optimizations are possible with automata-driven indexing. With these optimizations, the automata-driven indexing has been integrated into the ALS Prolog system. Use of this technique resulted in speed improvements of 0% (i.e., no performance degradation for any program) to 30% for typical programs. This implementation performs indexing in *multiple stages*. Each stage starts off with a filtered set of candidate terms, and uses the indexing approach implemented within the stage to further reduce the candidate set. Successive stages perform increasingly complex operations for indexing. In the *ALS* implementation, the first stage performed first-argument indexing, the second stage used the coarse filtering technique above, while the final stage performed the full-blown version of automata-driven indexing. Although this particular implementation required three stages to gain consistent performance improvement, it is possible to integrate the first and second stages without suffering performance penalty. Thus, speed improvements could be gained with this approach by just using the coarse-filtering stage before the full-blown automata-driven indexing stage.

8.3. Summary of advantages and disadvantages

Automata-driven indexing factors the operations involved in matching the function symbols of the query term with those from the indexed terms. In particular, it ensures that *no symbol in the query term is ever examined more than once.* This contrasts with some of the indexing techniques described earlier (e.g., discrimination trees) where symbols may have to be reexamined (potentially many times) due to backtracking. Although adaptive indexing avoids reexamination of symbols, this is achieved at the cost of a potential exponential blow-up in the size of the index, which is avoided in automata-driven indexing. On the negative side, note that this technique generates multiple preorder strings from each indexed term, similar to path indexing. Combining the results involving individual preorder strings (so as to determine compatibility of the indexed and query terms) is time-consuming, as we

[5]The resulting automaton would look like a discrimination tree for the indexed term, except every path in this tree is truncated at the first transition that is labelled with a '*'.

have to sequence through many indexed terms. Automata-driven indexing shares this drawback with path indexing, which may also end up spending a substantial amount of time sequencing through the indexed terms in the combination step. In practice, however, we find that the overhead of such combination steps is lower in the case of path indexing than automata driven indexing. On the other hand, automata driven indexing generates fewer strings from each indexed term than path indexing, and thus the number of combination steps is reduced.

In the worst-case, automata-driven indexing requires $O(|s| + \sum_{i=1}^{i=n} t_i)$ time to index n indexed terms $t_1, ..., t_n$. We note that the worst-case performance of automata-driven indexing cannot be improved upon in general. This is because in the worst case, all of the indexed terms are unifiable with a query term, and so we need to compute the substitutions for all of the variables in all of the indexed terms. This leads us to a lower bound that is the same as the runtime complexity of automata-driven indexing. In practice, however, techniques such as unification factoring are better suited for dealing with root-unifications.

It must be emphasized that a key benefit of automata-driven indexing is that the ideas are applicable to handle indexing questions that involve all of the subterms of the query term. None of the other indexing techniques discussed in this paper are able to reuse the efforts involved in unifying the query term at the root for operations involving unification of the subterms. The interested reader is referred to [Ramesh et al. 1994] for details.

9. Code trees

Code trees were introduced in [Voronkov 1994, Voronkov 1995]. A code tree is an index consisting of pieces of code instead of strings. Every piece of code is an instruction of an abstract machine able to perform the retrieval operation.

The general scheme of this indexing technique is as follows. Suppose that we have a retrieval condition R. For every indexed term l, we compile l into a sequence of instructions $i_1, \dots, i_n$ of the abstract machine. This sequence represents a function I_l such that for every possible query term t we have $I_l(t) \Leftrightarrow R(l, t)$. The sequence of instructions may, in general, not execute sequentially. For example, it may have branching or jump instructions.

Given a collection $\mathcal{L}$ of terms, the code tree for this collection is constructed as follows. For every $l \in L$ we compile the corresponding sequence of instructions representing I_l. Then we integrate this sequence of instructions into a (large) automaton called *code tree*. The code tree represents the function $I_{\mathcal{L}}$ such that $I_{\mathcal{L}}(t)$ returns the list of all $l \in \mathcal{L}$ such that $R(l, t)$.

Code trees are used in Vampire to implement retrieval of generalizations (used in *forward demodulation*) and multiliteral subsumption (see Section 13.2).

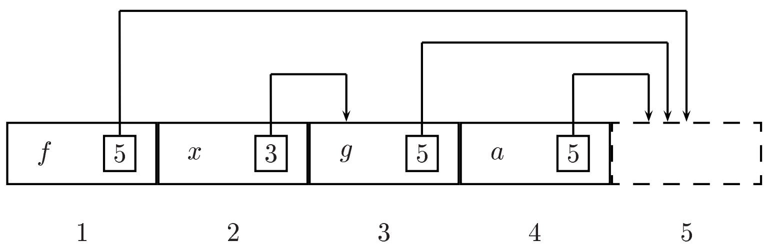


Figure 30: Flatterm structure for $f(x, g(a))$

9.1. Retrieval of generalizations

Algorithms for performing retrieval on different representations of a query term may differ in a number of details. For example, when we perform depth-first traversal of a query term $f(s,t)$ represented as a tree and go down from the symbol f to the subterm s we have to memorize the subterm t, since t should be traversed after the traversal of s has been completed. If we use the flatterm representation of query terms, memorizing t is unnecessary, since we will arrive at the term t anyhow when the traversal of s will have been completed.

Since a code tree represents a code for performing retrieval, the code contains instructions for performing traversal of the query term and thus the set of instructions used in code trees may depend on the chosen representation of query terms. In this section we choose the *flatterm* representation considered in *partially adaptive code trees*partially adaptive code treecode tree!partially adaptive [Riazanov and Voronkov 2000*b*]. A version of code trees working with nearly an arbitrary representation of query terms is presented in [Voronkov 1995].

We will use the term traversal functions $next_t$ and $after_t$ introduced in Section 6.2.2.

We denote by $|t|$ the *size* of a term t. Using flatterms, a term t is represented by an array of the size $|t| + 1$. Let $p_1 < \ldots < p_n$ be all positions in t. Then the i-th element of the array is a pair $\langle s, j\rangle$, where $s = root(t/p_i)$ and $p_j = after_t(p_i)$. For example, the term $f(x, g(a))$ is represented by the structure shown on Figure 30.

In can be seen that computation of our two major term traversal operations on positions, $next_t$ and $after_t$, can be done very efficiently on such a representation. $next_t$ is computed by a simple incrementation of the corresponding subscript, so $next_t(p_i) = p_{i+1}$, and the subscript of $after_t(p_i)$ is given in the ith element explicitly. Another serious advantage of this representation in comparison with tree-like terms is that equality of two subterms t/p_i and q/t_j can be checked efficiently, without using stack operations.

Let us fix a term l. Let $\Lambda = p_1 < p_2 < \ldots < p_n$ be all proper positions in l. Then for $i \in \{1, \ldots, n\}$, $pos_i(l)$ will denote p_i.

Let us give several definitions that will be used in the description of subsumption algorithms below. Let $p_{k_1} < \ldots < p_{k_m}$ be all variable positions in l. The *i-th variable position* in l, denoted by $vp_i(l)$, is defined as $vp_i(l) = p_{k_i}$. For $i > m$ $vp_i(l)$ is undefined. The *normalized form* of a term l, denoted by $norm(l)$, is the term obtained from l be replacing the subterm of l at the ith variable position by the variable $*_i$, for all i. For example, the normalized form of $f(x_1, a, g(x_1, x_2))$ is $f(*_1, a, g(*_2, *_3))$.

The *variable equivalence relation* for a term l, denoted $\mathcal{E}_t$, is the equivalence relation on $\{1, \ldots, m\}$ such that: $\langle i, j\rangle \in \mathcal{E}_t$ if and only if $root(t/vp_i(l)) = root(t/vp_j(l))$. For example, the variable equivalence relation for $f(x_1, a, g(x_1, x_2))$ consists of two equivalence classes: $\{1, 2\}$ and $\{3\}$. Note that two terms s, t are variants of each other if and only if the pair $\langle norm(s), \mathcal{E}_s\rangle$ coincides with the pair $\langle norm(l), \mathcal{E}_t\rangle$. If $\mathcal{B}$ is a binary relation, $\mathcal{B}^{\approx}$ denotes the transitive, reflexive and symmetric closure of $\mathcal{B}$. If $\mathcal{E}$ is an equivalence relation and $\mathcal{B}$ is such a binary relation that $\mathcal{B}^{\approx} = \mathcal{E}$, then $\mathcal{B}$ is called a *frame* of $\mathcal{E}$. A frame is called *minimal* if no proper subset of it is a frame. Throughout the rest of this paper we consider only equivalence relations over finite sets of the form $\{1, \ldots, m\}$. A finite sequence $\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle$ of pairs of integers is called a *computation sequence* for $\mathcal{E}$ if the relation $\{\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle\}$ is a minimal frame of $\mathcal{E}$ and $u_i < v_i$ for all $i \in \{1, \ldots, k\}$. Such a computation sequence is called *canonical* if each u_i is the minimal element of its equivalence class in $\mathcal{E}$ and for $i < j$ we have $\langle u_1, v_1\rangle <_{lex} \ldots <_{lex} \langle u_k, v_k\rangle$, where $<_{lex}$ is the standard lexicographic (i.e., componentwise) order on integers. Note that the canonical computation sequence is uniquely defined.

Consider an example: equivalence relation consisting of two equivalence classes: $\{1, 3, 5, 7\}$ and $\{2, 4\}$. The canonical computation sequence for this relation is $\{\langle 1, 3\rangle, \langle 1, 5\rangle, \langle 1, 7\rangle, \langle 2, 4\rangle\}$. Another computation sequence for this equivalence relation is $\{\langle 1, 3\rangle, \langle 3, 5\rangle, \langle 3, 7\rangle, \langle 2, 4\rangle\}$.

9.2. Compilation for forward subsumption by one term

In order to define instructions of an abstract machine for performing subsumption, we will first define a subsumption algorithm. On unit clauses, subsumption is equivalent to matching. We are going to solve the following problem: given a term l and a query term t we have to check if l subsumes t. Figure 31 shows a deterministic algorithm that implements forward subsumption.

Following [Voronkov 1995] we specialize this general subsumption algorithm *Subsume* for each indexed term l, obtaining its specialized version $Subsume_l$. The specialized version has the property $Subsume_l(t) = Subsume(l, t)$, for each query term t. The specialized algorithm is represented as a sequence of instructions of an abstract machine. In other words, we *compile* the term into code of the abstract machine. Then this code is submitted, together with the query term t, to the interpreting procedure. Before presenting technical details, let us consider a simple example.

```
procedure Subsume(l, t)
begin
  /* First phase: term traversal */
  let subst be an array for storing positions in l;
  pos_l := Λ;
  pos_t := Λ;
  while pos_l ≠ ε
    if norm(l)/pos_l = *_i then
      subst[i] := pos_t;
      pos_t := after_t(pos_t);
      pos_l := after_l(pos_l);
    else /* l/pos_l is not a variable */
      if root(l/pos_l) ≠ root(t/pos_t) then
        return failure;
      else
        pos_t := next_t(pos_t);
        pos_l := next_l(pos_l);
      endif;
    endif;
  end while;
  /* Second phase: comparison of terms */
  let ⟨u_1, v_1⟩, ..., ⟨u_n, v_n⟩ be the canonical computation sequence for ℰ_l.
  forall i ∈ {1, ..., n}
    if t/subst[u_i] ≠ t/subst[v_i] then return failure;
  end forall
  return success;
end
```

Figure 31: A term subsumption algorithm

9.1. EXAMPLE. Let $l = f(g(x_1, x_2), h(x_1, x_1))$ be an indexed term. The specialised version of the matching algorithm for this term is shown in Figure 32.

This specialized version can be rewritten in a more formal way using special instructions *Initialize*, *Check*, *Put*, *Compare*, *Success* and *Failure* as shown in Figure 33. The semantics of these instructions should be clear from the example, but will also be formally explained later.

9.3. Abstract subsumption machine

Now we are ready to describe the abstract machine, its instructions, compilation process, and interpretation formally. Memory of the abstract machine is divided into the following "registers":

1. substitution register *subst* which is an array of positions in the query term;
2. register p for storing the current position in the query term;
3. a register *instr* for storing the label of the current instruction.

procedure $Subsume_l(t)$
begin
  $p := \Lambda$;
  if $root(t/p) \neq f$ return failure;
  $p := next_t(p)$;
  if $root(t/p) \neq g$ return failure;
  $p := next_t(p)$;
  $subst[1] := p$;
  $p := after_t(p)$;
  $subst[2] := p$;
  $p := after_t(p)$;
  if $root(t/p) \neq h$ return failure;
  $p := next_t(p)$;
  $subst[3] := p$;
  $p := after_t(p)$;
  $subst[4] := p$;
  $p := after_t(p)$;
  if $t/subst[1] \neq t/subst[3]$ return failure;
  if $t/subst[1] \neq t/subst[4]$ return failure;
  return success;
end

Figure 32: The algorithm *Subsume* specialized for the term $l = f(g(x_1, x_2), h(x_1, x_1))$

$initl$: $Initialize(m_1)$
m_1 : $Check(f, m_2, faill)$
m_2 : $Check(g, m_3, faill)$
m_3 : $Put(1, m_4, faill)$
m_4 : $Put(2, m_5, faill)$
m_5 : $Check(h, m_6, faill)$
m_6 : $Put(3, m_7, faill)$
m_7 : $Put(4, m_8, faill)$
m_8 : $Compare(1, 3, m_9, faill)$
m_9 : $Compare(1, 4, m_{10}, faill)$
m_{10} : $Success$
$faill$: $Failure$

Figure 33: The corresponding sequence of instructions

To identify instructions in code we will use *labels*. We distinguish two special labels: *initl*, and *faill*. A *labeled instruction* will be written as a pair of the form $m : I$, where m is a label and I is the instruction itself. The instruction set of our abstract machine consists of *Initialize*, *Check*, *Put*, *Compare*, *Success* and *Failure*. *Success* and *Failure* have no arguments. Other instruction have the following form:

- $Initialize(m_1)$, where m_1 is a label;
- $Check(f, m_1, m_2)$, where f is a function symbol and m_1, m_2 are labels;
- $Put(n, m_1, m_2)$, where n is a positive integer and m_1, m_2 are labels;
- $Compare(n_1, n_2, m_1, m_2)$, where n_1, n_2 are positive integers and m_1, m_2 are labels.

For convenience, we define two functions on instructions, *cont* and *back*. On all the above instructions *cont* returns m_1 and *back* returns m_2. Intuitively, *cont* is the label of the instructions that should be executed after the current instruction (if this instruction succeeds), and *back* is the label of the instruction that is executed if the current instruction fails.

The semantics of the instructions is shown in Figure 34. At the moment the last argument of *Put* is dummy. It will be used when we discuss the case of many

$Initialize(m_1)$	$p := \Lambda$; goto m_1
$Check(s, m_1, m_2)$	if $root(t/p) = s$ then $p := next_t(p)$; goto m_1 else goto m_2
$Put(n, m_1, m_2)$	$subst[n] := p$; $p := after_t(p)$; goto m_1
$Compare(m, n, m_1, m_2)$	if $t/subst[m] = t/subst[n]$ then goto m_1; else goto m_2
$Success$	return success
$Failure$	return failure

Figure 34: Semantics of instructions in code sequences

indexed terms.

For a given indexed term l, compilation of instructions for $Subsume_l$ results in a set of labeled instructions, called the *code for* l. It consists of two parts: *traversal code* and *compare code* plus three standard instructions: $initl$: $Initialize(m_1)$, $succl$: $Success$ and $faill$: $Failure$.

Suppose $p_1 < p_2 < \ldots < p_k$ are all positions in l. The *traversal code* for l is the set of labeled instructions $\{m_1 : I_1, \ldots, m_k : I_k\}$, where m_i's are labels and I_i's are defined as follows:

$$I_i = \begin{cases} Check(root(l/p_i), m_{i+1}, faill), & \text{if } l/p_i \text{ is not a variable;} \\ Put(k, m_{i+1}, faill), & \text{if } norm(l)/p_i = *_k. \end{cases}$$

Let $\langle u_1, v_1\rangle, \ldots, \langle u_n, v_n\rangle$ be the canonical computation sequence for $\mathcal{E}_l$. Then the *compare code* for l is the set of instructions m_{k+i} : $Compare(u_i, v_i, m_{k+i+1}, faill)$ for $i \in \{1, \ldots, n\}$, where $m_{k+n+1} = succl$. In Figure 33 from example 9.1 instructions m_1–m_7 and m_8, m_9 form the traversal and the compare code respectively.

The code for l is executed on the query term according to the semantics of instructions shown in Figure 34, beginning with the instruction *Initialize*. The following statement is unlikely to surprise anybody: execution of the code for l on any query term t terminates and returns *success* if and only if l subsumes t. Observe that code for l has a linear structure: instructions can be executed sequentially. In view of this observation we will call code for l also the *code sequence* for l.

$initl : Initialize(m_1)$	$initl : Initialize(m_1)$
$m_1 : Check(f, m_2, faill)$	$m_1 : Check(f, m_2, faill)$
$m_2 : Check(f, m_3, faill)$	$m_2 : Check(f, m_3, faill)$
$m_3 : Put(1, m_4, faill)$	$m_3 : Put(1, m_4, faill)$
$m_4 : Put(2, m_5, faill)$	$m_4 : Put(2, m_5, faill)$
$m_5 : Check(f, m_6, faill)$	$m_5 : Check(f, m_6, faill)$
$m_6 : Put(3, m_7, faill)$	$m_6 : Put(3, m_7, faill)$
$m_7 : Put(4, m_8, faill)$	$m_7 : Put(4, m_8, faill)$
$m_8 : Compare(1, 3, m_9, faill)$	$m_8 : Compare(2, 3, m_9, faill)$
$m_9 : Compare(1, 4, m_{10}, faill)$	$m_9 : Compare(2, 4, m_{10}, faill)$
$m_{10} : Success$	$m_{10} : Success$

Figure 35: Code sequences for two terms

9.4. Code tree for a set of indexed terms

Recall that our main problem is to find if any term l in a large set L of indexed terms subsumes a given query term t. Using compilation described in the previous subsection, one can solve the problem by the execution of the codes for all terms in L. This solution is inappropriate for large sets of terms. However, code sequences for terms can still be useful as we can share many instructions from code for different terms. We rely on the following observation: in most instances in automated theorem proving the set L contains many terms having similar structure. Code sequences for similar terms often have long coinciding prefixes. It is natural to combine the code sequences into one indexing structure, where the equal prefixes of code sequences are shared. Due to the tree-like form of such structures we call them *code trees*. Nodes of code trees are instructions of the abstract subsumption machine. Linking of different code sequences is done by setting appropriate values to the *cont* and *back* arguments of the instructions. A branch of such a tree is a code sequence for some indexed term interleaved by some instructions of code sequences for other indexed terms. Apart from reducing memory consumption, combining code sequences in one index results in tremendous improvements in time-efficiency, since during a subsumption check shared instructions are executed once for several terms in the indexed set. To illustrate this idea, let us compare the code sequences for the terms $l_1 = f(f(x_1, x_2), f(x_1, x_1))$ and $l_2 = f(f(x_1, x_2), f(x_2, x_2))$ given in Figure 35.

Sharing the first eight instructions of this results in the code given in Figure 36. This figure also illustrates control flow of this code.

We can execute this code as follows. First, the eight shared instructions are executed. If none of them results in failure, we continue by executing instructions m_8, m_9, m_{10}. If the *Success* instruction m_{10} is reached the whole process terminates with success. Otherwise, if any of the equality checks m_8, m_9, failed, we have to backtrack and resume the execution from the instruction m_{11}.

In general, to maintain a code tree $C_{\mathcal{L}}$ for a dynamically changing set $\mathcal{L}$, one has to implement two operations: integration of new code sequences into the tree, when

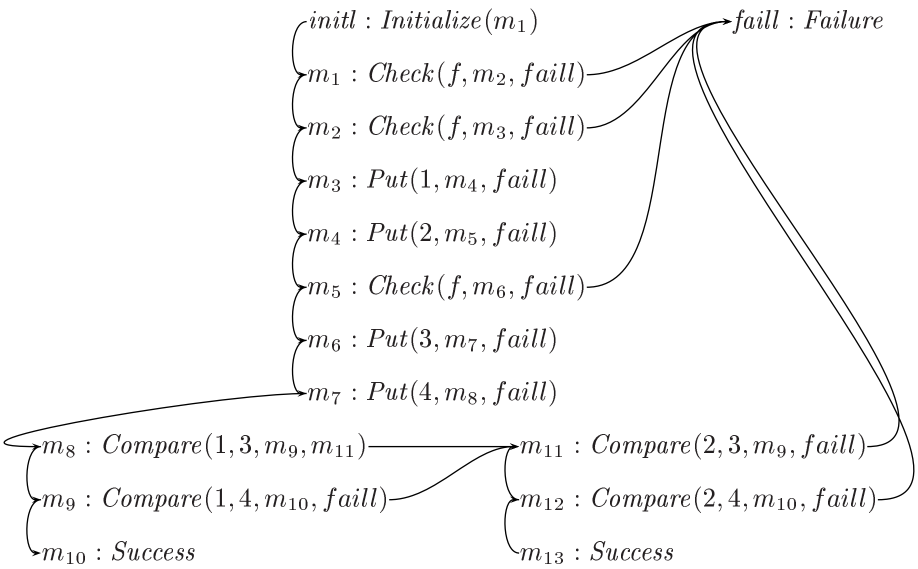


Figure 36: Code tree for two terms

a term is inserted in $\mathcal{L}$, and removal of sequences when a term is deleted from $\mathcal{L}$. The integration of a code sequence C_l into a code tree $C_{\mathcal{L}}$ can be done as follows. We move simultaneously along the sequence C_l and a branch of $C_{\mathcal{L}}$ beginning from the *Initialize* instructions. If the current instruction $I_{\mathcal{L}}$ in $C_{\mathcal{L}}$ coincides with the current instruction I_l in C_l up to the label arguments, we follow down the instructions in their *cont* arguments. If $I_{\mathcal{L}}$ differs from I_l we have to consider two cases:

1. If $back(I_{\mathcal{L}})$ is not the *Failure* instruction, then in the code tree we move to this instruction and continue integration.
2. If $back(I_{\mathcal{L}})$ is *Failure*, we set the *back* argument of $I_{\mathcal{L}}$ to the label of I_l. Thus, the rest of the code sequence C_l together with the passed instructions in C_l forms a new branch in the tree.

Removal of obsolete branches is also very simple: we remove from the code all unshared instructions corresponding to the removed term and link the remaining instructions in an appropriate manner. Due to postponing *Compare* instructions, code trees maintained in this manner have an important property: traversal codes for any terms having the same normalized form are shared completely.

Code trees are executed nearly the same way as code sequences, but with one difference due to possible backtrack points. As soon as an instruction with a backtrack argument is found, we store its backtrack argument and the current position in the query term in special stacks *backtrPos* and *backtrInstr*. Semantics of instructions in code trees is shown in Figure 37.

$Initialize(m_1)$	$p := \Lambda$; $backtrPos := empty\ stack$; $backtrInstr := empty\ stack$; goto m_1

$Check(s, m_1, m_2)$	if $root(t/p) = s$ then
	  $push(m_2, backtrInst)$;
	  $push(p, backtrPos)$;
	  $p := next_t(p)$;
	  goto m_1
	else goto m_2

$Put(n, m_1, m_2)$	$push(m_2, backtrInst)$; $push(p, backtrPos)$; $subst[n] := p$; $p := after_t(p)$; goto m_1

$Compare(k, n, m_1, m_2)$	if $l/subst[k] = l/subst[n]$ then
	  $push(m_2, backtrInst)$;
	  $push(p, backtrPos)$;
	  goto m_1
	else goto m_2

Success	return success
Failure	if $backtrPos$ is empty then return failure $p = pop(backtrPos)$; goto $pop(backtrInst)$

Figure 37: Semantics of instructions in code trees

It is worth noting that all operations in the semantics of the instructions can be executed very efficiently on flatterms. Riazanov and Voronkov [2000*b*] consider *partially adaptive code trees*, in which the *Compare* instruction do not necessarily correspond to the canonical sequence for $\mathcal{E}_l$. Moreover, these instructions can be moved up and down the code tree.

The experiments described in Nieuwenhuis, Hillenbrand, Riazanov and Voronkov [2001] have shown that, for retrieval of generalization, code trees (as implemented in VAMPIRE) are faster than perfect discrimination trees (as implemented in WALDMEISTER) by about a factor of 1.4 and faster than context trees implemented in FIESTA by about a factor of 1.9. Code trees use about 1.2 times more space compared to context trees and about 4.6 time less space than discrimination trees.[6]

10. Substitution trees

10.1. Overview

Substitution trees [Graf 1995] extend the model of indexing presented earlier so that comparisons in the index no longer involve simple tests of equality on nonvariable symbols, but can test for unifiability among terms. This is achieved by storing substitutions rather than terms or p-strings. The idea of using arbitrary unification operations in the index can be traced to abstraction trees [Ohlbach 1990]. As a result of this, substitution trees can be smaller in size. Moreover, substitution trees can factor out the computation of substitutions, as opposed to just matching operations involving nonvariable symbols.

We will use *normalized variables* in the indexed terms as defined on page 1889. For instance, we rename the indexed terms $f(x, a)$ into $f(x_1, a)$ and $f(x_1, x_2)$. In addition to the normalized variables $x_1, x_2, \dots$ we will use a sequence of variables $*_0, *_1, \dots$, disjoint from the variables of indexed or query terms, to represent substitutions. Substitution of a term t for a variable $*_i$ will be denoted by an equality $*_i = t$. In substitution trees, instead of storing a term t, we store a substitution $*_0 = t$ represented as a composition of substitutions for $*_i$. For example, the term $f(g(a, x_1))$ can be stored as a composition of such substitutions in several different ways, including $*_0 = f(g(a, x_1))$ and $*_0 = f(g(*_1, x_1)),\ *_1 = a$. Substitution trees share common parts of substitutions rather than common prefixes. Every branches in a substitution tree represents an indexed term, obtained by composing the substitutions on this branch and applying the resulting substitution to $*_0$.

10.1. Example. We illustrate substitution tree indexing with an example set consisting of four indexed terms

$$\begin{array}{ll}(1)\ f(x_1, x_1), & (2)\ f(x_1, x_2),\\ (3)\ f(a, g(d)), & (4)\ f(g(d), g(x_1)).\end{array}$$

in Figure 38. By composing the substitutions on e.g., the rightmost branch:

$$*_0 = f(*_2, *_1),\ *_1 = g(*_3),\ *_2 = g(d),\ *_3 = x_1,$$

we obtain the substitution of $f(g(d), g(x_1))$ for $*_0$ representing indexed term 4.

[6] Waldmeister's discrimination trees are array-based, so every node occupies a considerable amount of memory, even if only one term is stored in this node.

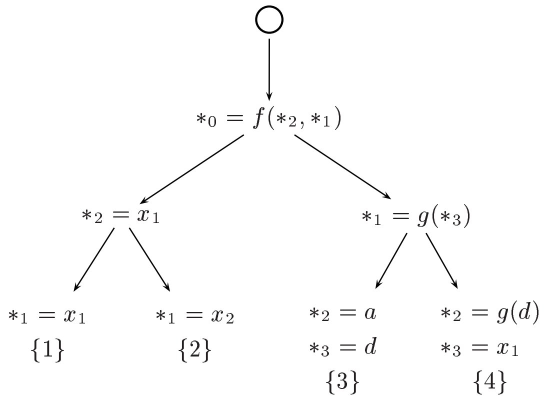


Figure 38: A substitution tree

10.2. Index maintenance

Before discussing indexing algorithms, let us note one feature of substitution trees: the order of term traversal is not fixed in advance. For example, for the substitution tree of Figure 38, the substitution for the first argument of f is done before the substitution for its second argument in indexed terms $1, 2$, but it is done after in indexed terms $3, 4$. So when we traverse indexed terms $1, 2$, we traverse the arguments of f left-to-right, while for indexed terms $3, 4$ we traverse them right-to-left. This feature may lead to very compact substitution trees, but it also has some undesirable consequences for the indexing algorithms:

1. There may be several different ways to *insert* a term in a substitution tree. For example, if we insert $f(x_1, g(x_2))$ in the substitution tree of Figure 38, we may follow down any of the transitions coming out from $*_0 = f(*_1, *_2)$. If we follow the left one, we share the substitution $*_1 = x_1$, if we follow the right one, we share $*_2 = g(x_3)$. This property can be used to find an optimal way of inserting a term and lead to even more compact substitution trees, but optimal insertion requires more complex algorithms.
2. When we *delete* a term t from a substitution tree, if there are several transitions coming out from a node, we cannot decide which transition corresponds to t by simply looking at the children of this node. Therefore, algorithms for deletion of a term from a substitution tree involve some kind of backtracking.
3. *Retrieval* may result in a larger amount of backtracking steps compared to other indexing techniques. For example, if all of the indexed terms and the query term are ground, retrieval using all previously studied indexing techniques will be deterministic, but retrieval using substitution trees may require backtracking even in this case.

In this paper we present a version of substitution trees called *linear substitution trees* of Graf [1996]. In linear substitution trees, on any root-to-leaf path, each variable $*_i$ occurs at most once in right-hand sides of substitutions. For example, the substitution $*_0 = f(*_1, *_1)$ cannot occur in a linear substitution tree. Likewise, two substitutions $*_1 = f(*_3)$ and $*_2 = g(*_3)$ cannot occur on the same branch. However, the substitution $*_0 = f(x_1, x_1)$ is perfectly legal.

Insertion of indexed terms $l_1, ..., l_n$ is viewed as insertion of the substitutions $*_0 = l_1, ..., *_0 = l_n$. The insertion process works by following down a path in the tree that is *compatible* with the substitution ρ to be inserted. To formally define insertion and deletion of substitutions, let us introduce a few notions.

Let l_1, l_2 be terms and V be a set of variables $\{*_i, *_{i+1}, *_{i+2}, \ldots\}$ for some $i \geq 0$, such that V is disjoint from the variables of l_1, l_2. The *most specific linear common generalization* of l_1 and l_2 with respect to V, denoted $mslcg(l_1, l_2, V)$ is defined as follows.[7]

1. If either l_1 or l_2 is a variable $*_k$, then $mslcg(l_1, l_2)$ is $*_k$.
2. Otherwise, if l_1 coincides with l_2, then $mslcg(l_1, l_2) = l_1$.
3. Otherwise, let p be the first (in the preorder traversal order) position in both l_1 and l_2 such that the top symbols of l_1/p and l_2/p are different. Consider two cases
   (a) Either l_1/p or l_2/p is a variable $*_k$. Define $l'_1 = l_1[*_k]_p$ and $l'_2 = l_2[*_k]_p$. Then define $mslcg(l_1, l_2, V)$ to be the most specific linear common generalization of l'_1 and l'_2 with respect to V.
   (b) Otherwise, define $l'_1 = l_1[*_i]_p$ and $l'_2 = l_2[*_i]_p$. Then define $mslcg(l_1, l_2, V)$ to be the most specific linear common generalization of l'_1 and l'_2 with respect to $V - \{*_i\}$.

For example, the most specific linear common generalization of $f(g(x_1), g(x_1))$ and $f(x_1, g(x_2))$ with respect to $\{*_2, \ldots\}$ is the term $f(*_2, g(*_3))$. Likewise, the most specific linear common generalization of $f(g(x_1), g(*_1))$ and $f(x_1, g(x_2))$ with respect to $\{*_2, \ldots\}$ is the term $f(*_2, g(*_1))$.

Let σ_1 and σ_2 be two substitutions and V be a set of variables such that both V and the domains of σ_1 and σ_2 are subsets of $\{*_0, *_1, \ldots\}$. The *most specific linear common generalization* of σ_1 and σ_2 with respect to V, denoted $mslcg(\sigma_1, \sigma_2, V)$, is defined as follows. Consider the set X of all variables x such that $x\sigma_1$ and $x\sigma_2$ have the same top symbol and $x\sigma_1 \neq x$. Let $X = \{*_1, \ldots, *_j\}$ such that $i < \ldots < j$. Take any function symbol h and let s be the most specific linear common generalization of the terms $h(*_i\sigma_1, \ldots, *_j\sigma_1)$ and $h(*_i\sigma_2, \ldots, *_j\sigma_2)$ with respect to V. Then s has the form $h(s_i, \ldots, s_j)$ for some terms $s_i, \ldots, s_j$. Define $mslcg(\sigma_1, \sigma_2, V)$ to be the substitution $\{*_i = s_i, \ldots, *_j = s_j\}$.

10.2. EXAMPLE. The most specific linear common generalization of the substitutions

$$\sigma_1 = \{*_1 = g(a), *_2 = f(c, x_1), *_3 = g(c)\}$$
$$\sigma_2 = \{*_1 = g(b), *_2 = x_1, *_3 = g(x_2)\}$$

[7] Our notion of most specific linear common generalization is slightly nonstandard, since we treat variables in $\{*_0, *_1, \ldots\}$ differently from other variables. However, one can prove that the result is always the most general linear term that generalizes both l_1 and l_2.

```
function insert(node n, substitution σ)
if ∃ a child n' of n labelled by ρ such that ρ is compatible with σ then
    τ := mslcg(σ, ρ);
    if τ = ρ then
        insert(n', σ/ρ)
    else
        change label of n' to ρ/τ;
        insert a new node n'' labelled by τ between n and n';
        add to n'' as a child a new leaf labelled by σ/τ
    endif
else
    add to n as a child a new leaf labelled by σ
endif
```

Figure 39: Algorithm for insertion into a substitution tree

with respect to $\{*_4, \ldots\}$ is computed as follows. First, the set X consists of the variables $*_1$ and $*_3$ because σ_1 and σ_2 disagree on the top symbols of $*_2$. Then we have to compute the most specific linear common generalization of $h(g(a), g(c))$ and $h(g(b), g(x_2))$, that is $h(g(*_4), g(*_5))$. Therefore, the most specific linear common generalization of σ_1 and σ_2 is the substitution $\rho = \{*_1 = g(*_4), *_3 = g(*_5)\}$.

Two substitutions σ_1 and σ_2 are called *compatible* if their most specific linear common generalization is nonempty. Let σ and ρ be two substitutions. If there exists a substitution τ such that for every variable x in the domain of σ we have $x\rho\tau = x\sigma$, we denote τ by σ/ρ. It is not hard to argue that for any compatible substitutions σ_1, σ_2 and their most specific linear common generalization ρ, both σ_1/ρ and σ_2/ρ are defined. For instance, for the substitutions of Example 10.2, we have

$$\sigma_1/\rho = \{*_2 = f(c, x_1), *_4 = a, *_5 = c\};$$
$$\sigma_2/\rho = \{*_2 = x_1, *_4 = b, *_5 = x_2\}.$$

An algorithm for insertion in substitution trees is given in Figure 39. To insert a term l in the substitution tree, one should call the insertion function *insert* using, as arguments, the top node of the tree and the substitution $*_0 = l$.

We will illustrate this algorithm on the terms used in Example 10.1, assuming that they are inserted in the order of their numbers. The substitution tree containing all four terms was already shown in Figure 38. The intermediate substitution trees consisting of the terms (1), (1)–(2), and (1)–(3) are shown in Figure 40. In general, a different order of inserting indexed terms in a substitution tree may result in different trees.

Deletion of a term l from a substitution tree is quite straightforward except that backtracking is required to find the leaf corresponding to the substitution $*_0 = l$

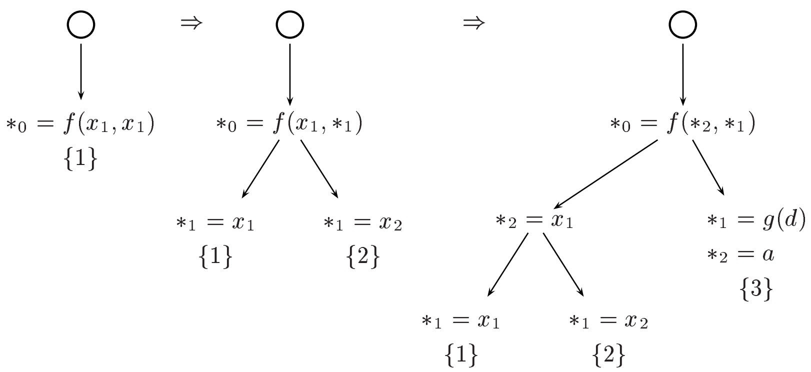


Figure 40: Insertion of terms into a substitution tree

to be deleted. When the leaf is found, we delete all nodes that correspond only to this substitution and repair the tree, by collapsing sequences of nodes with a single child into one node. For example, deletion of term (4) from the substitution tree of Figure 38 results in the rightmost substitution tree among those in Figure 40.

10.3. Retrieval of unifiable terms

When an indexing technique is used as a perfect filter, some retrieval conditions may be more difficult to handle than others. For example, for discrimination trees retrieval of generalizations is straightforward, but retrieval of unifiable terms of instances is more difficult to implement because of embedded variables.

Substitution trees differ from other indexing techniques in this aspect: all retrieval operations have quite a straightforward implementation on substitution trees. As a result, some provers (Spass and Fiesta) use substitution trees as a single indexing data structure. This feature is due to storing substitutions rather than symbols at nodes. The price to pay is that an operation performed at visiting a node is not a simple comparison of symbols, but may involve complex operations, like unification. We will demonstrate this by retrieval of unifiable terms from substitution trees.

Retrieval proceeds by following down all paths in the substitution tree that contain only substitutions compatible with the given query term. Specifically, we follow down each edge in the substitution tree, starting from the root, until we hit a leaf or we reach a substitution that is incompatible with the query term. In the former case, the indexed terms associated with the leaf are added to the candidate set, whereas in the latter case we prune the search operation at this node. It is convenient to perform this operation using a backtracking algorithm, as with discrimination trees. As an example, consider the substitution tree of Figure 38 and the query term $f(f(a, y_1), y_1)$.

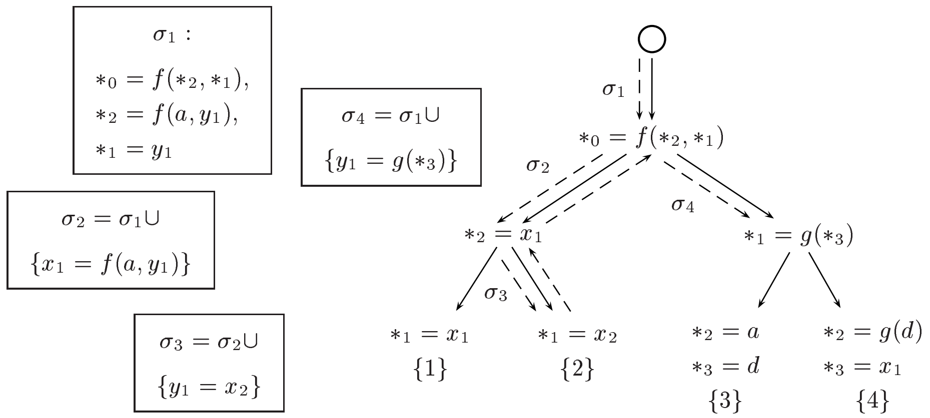


Figure 41: Retrieval of terms unifiable with $f(f(a, y_1), y_1)$ from a substitution tree

Retrieval is demonstrated in Figure 41. We use dashed arrows to show the nodes visited successfully. The computed substitutions are illustrated at the left-hand side of the picture.

Graf [1996] describes some variations of substitution trees. Here we presented linear substitution trees, but they can also be nonlinear, containing substitutions like $*_0 = f(*_1, *_1)$. In *weighted substitution trees* we maintain additional information about size of substitutions. This information can be used to prune some of the unsuccessful paths in the tree very quickly, rather than proceeding until a point where an incompatible substitution is seen.

11. Context trees

Context trees is a data structure for indexing introduced only recently by Ganzinger, Nieuwenhuis and Nivela [2001]. Since we learned about this technique just a couple of weeks before this volume goes to print, we present it here only sketchily.

Context trees generalize of substitution trees so that one can use variables ranging over function symbols. For example, the term $f(a, b)$ can be represented using a composition of substitutions $*_0 = *_1(a, b), *_1 = f$. Thus, terms with different function symbols can be shared, too.

Consider an example taken from [Ganzinger et al. 2001]. The set of indexed terms consists of the following three terms:

$$(1)\ h(x_1, f(s)), \qquad (2)\ h(x_1, g(s)), \qquad (3)\ h(b, f(s)).$$

where s is an arbitrary term. Figure 42 shows a left-to-right substitution tree (i.e., the traversal order is always left-to-right) and a context tree for this set of terms.

Context trees are more compact than substitution trees. Retrieval time depends on the query term. For example, for the query term $h(b, f(r))$ and unification as the

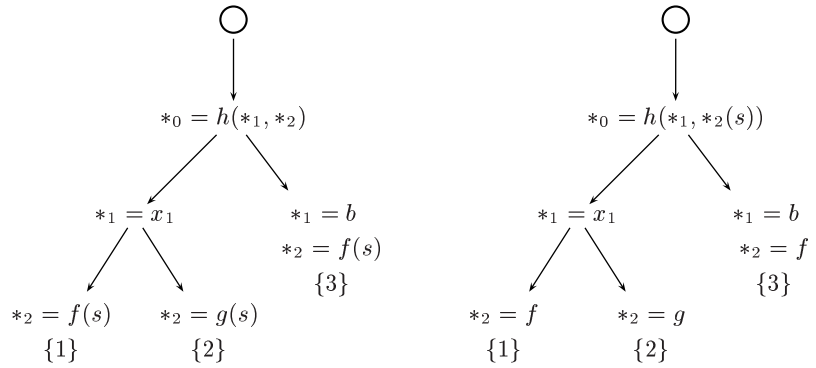


Figure 42: A substitution tree and a context tree

retrieval condition the term r will be unified three times with s in the substitution tree, and only once in the context tree of Figure 42. On the contrary, for the query term $h(b, k(r))$ the terms r and s will be unified once in the context tree, but no unification or r against s will take place in the substitution tree.

Context trees can be made even more compact when function symbols of different arities can be shared as well. The technique for this is presented in [Ganzinger et al. 2001] based on the idea of *curried terms*. Essentially, if we want to share the top function symbols in the terms $f(a, b)$ and $h(a)$, we introduce a new function symbol $\cdot$ to denote function application, and consider the terms with their arguments reversed, i.e., $(f \cdot b) \cdot a)$ and $h \cdot a$. Details can be found in [Ganzinger et al. 2001].

12. Unification factoring

12.1. Overview

Unification factoring [Dawson, Ramakrishnan, Ramakrishnan, Sagonas, Skiena, Swift and Warren 1995] is an indexing technique that is similar to substitution trees, but developed independently in the context of logic programming. It shares most of the advantages of substitution trees, including the ability to factor out substitutions. In addition, Dawson, Ramakrishnan, Ramakrishnan, Sagonas, Skiena, Swift and Warren [1995] studied *optimal* algorithms for construction of factoring automata. In this section, we provide a brief description of unification factoring, followed by a discussion of the optimality results.

Unification factoring was developed in the context of logic programming, where programs consists of a collection of *predicate definitions*. Each predicate definition is made up of *clauses* of the form

$$Head\ \texttt{:-}\ Body.$$

The set of indexed terms consist of all clause heads that define a single predicate,[8] and the retrieval condition is unifiability.

12.2. Complexity of constructing optimal automata

As with substitution trees, there are many possible ways to build an index, each with a differing performance. The interesting problem then is the design of *optimal* unification factoring that minimizes the cost of retrieval. One possible measure of cost is the number of edges traversed to perform a retrieval operation. This measure is reasonable if the cost of all unification operations in the index are of the same order. In the presence of nonlinear indexed terms, this can no longer be assured: note that the time required to perform an operation such as $X = Y$ will depend on the actual terms appearing in the query term in place of X and Y. As such, we limit our discussion to only linear indexed terms in this section, which will ensure that we never have to perform operations that require us to compare arbitrarily large terms.

Although counting the number of edges traversed appears to be a natural way to measure cost, it suffers from the drawback that the cost becomes a function of the query term. This makes it difficult to talk about cost of the automaton as a whole. In order to develop a measure that is independent of the query term, we can make use of the worst-case cost, which corresponds to a query term that has a single nonvariable symbol at its root, with all arguments being distinct variables. (This is the worst case since such a term will unify with every indexed term.) It has been shown [Dawson, Ramakrishnan, Ramakrishnan, Sagonas, Skiena, Swift and Warren 1995] that construction of optimal index is very hard in the case of *nonsequential automata* that deal with indexed terms with no priorities. They also present a polynomial time algorithm for optimal *sequential automata* that deal with indexed terms that are totally ordered in priority. Such sequential automata capture Prolog's strategy of sequentially trying the clauses defining a predicate. The complexity is also dependent on the availability of *mode* information, which specifies in advance (i.e., at index construction time) whether certain subterms in the query term will always be ground terms or always be (free) variables.

[8] The idea is to build different indexes for different predicates.

	Sequential	*Nonsequential*
Without modes	P (dynamic programming)	NP-complete
With modes	NP-hard in Π_2^p	NP-hard

12.3. Construction of optimal automata

We will annotate each state in the sequential unification factoring automaton (SFA) with (a) the prefix of the query term that would be inspected on the path from the root to that state, and (b) the set of indexed terms that are compatible with that prefix. We can treat the compatible set as a sequence, which would allow us to capture the (totally ordered) priority information. The following three properties are used in the construction of an optimal SFA.

1. In an SFA, the transitions from a state partition its compatible set into *subsequences.*
2. In an optimal SFA for a sequence of terms, states and transitions specifying unification operations common to the entire sequence form a "chain" (i.e., a sequence of nodes, each of which has only one outgoing transition).
3. Each subautomaton of an optimal SFA is itself optimal.

For the last point, we note that each sub-SFA will be associated with a different prefix at its root state which reflects the symbols that have been inspected already in reaching that state from the root of the entire SFA. We will hence talk about SFA as being parametrized with respect to the prefix and the compatible set (sequence) associated with its root state.

The construction of an optimal automaton is a recursive process in which, at each state starting with the root, the automaton is expanded based on the compatible set $\mathcal{C}$ of indexed terms and the prefix u associated with that state. We consider each position in the fringe of u, and select a position p such that costs of the optimal sub-SFA's created after inspecting p are minimized. From the properties of optimal SFA stated above, it follows that an SFA constructed in this manner will be optimal. The recursive construction process lends itself to a dynamic programming solution as described below. As mentioned earlier, we deal only with linear indexed terms.

We use a function *part* to partition the compatible sequence $\mathcal{C}$ into the minimum number of subsequences that share unification operations at this position.

12.1. DEFINITION *(Partition).* Given a sequence $\mathcal{C} = \langle l_1, \ldots, l_n \rangle$ of terms and a position π, the *partition of* $\mathcal{C}$ *by* π, denoted $part(\mathcal{C}, \pi)$, is the set of all triples (a, j, j'), $1 \leq j \leq j' \leq n$, such that all $\langle l_j, \ldots, l_{j'} \rangle$ is a maximal subsequence of $\langle l_1, \ldots, l_n \rangle$ having the same symbol a at the position π.

For example, the partition of the sequence $\langle p(a,a), p(a,b), p(b,c), p(a,d)\rangle$ at position 1 is the set $\{(a,1,2),(b,3,3),(a,4,4)\}$ corresponding to the following three maximal subsequences

$$\langle p(a,a), p(a,b)\rangle, \quad \langle p(b,c)\rangle, \quad \langle p(a,d)\rangle.$$

Each triple (a,j,j') computed by *part* corresponds to a transition from an SFA state with prefix u and compatible sequence $\mathcal{C}$. Note that all terms in the subsequence $\langle l_j,\ldots,l_{j'}\rangle$ possess identical symbols at the position p and possibly some of the other positions. Specifically, they are identical in all nonvariables in $mslcg(l_j,\ldots,l_{j'})$. It is clear from the above properties of optimal SFA's that the unification operations corresponding to the common positions (which are not already present in u) will be shared by the indexed terms in the subsequence. Thus, we can build an optimal SFA by identifying the choice of p that minimizes the cost (as per the equation given below), computing the partitions of $\mathcal{S}$ with respect to this position, introducing transitions corresponding to each subsequence $\langle l_j,...,l_{j'}\rangle$, and then using a recursive procedure to complete the sub-SFA reached by these transitions. Observe that the prefix corresponding to the new state reached by the transition (a,j,j') will be $mslcg(l_j/p,\ldots,l_{j'}/p)$. Therefore, we can compute the cost of a minimal SFA for a subsequence $\langle l_i,...,l_{i'}\rangle$ using the formula

$$cost(i,i',u) = \min_{\pi\in\mathcal{P}_v(u)} \left(\sum_{\substack{(a,j,j')\in\\ part(\langle l_i,\ldots,l_{i'}\rangle,\pi)}} (cost(j,j',u') + |u'| - |u|) \right), \qquad (12.1)$$

where u' stands for $mslcg(l_j,...,l_{j'})$, and $|t|$ denotes the number of nonvariable symbols in t. It is easy to see that the third parameter u of $cost(i,i',u)$ in the above equation is always $mslcg(l_i,...,l_{i'})$, and is hence uniquely determined by i and i'. Thus, we can eliminate u from the above equation, and compute the minimal SFA using a dynamic programming technique using a two-dimensional table indexed by i and i' for all values $1 \le i \le i' \le n$.

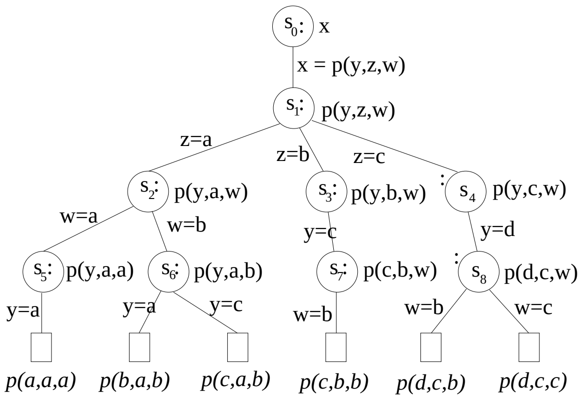
12.2. Example. Construction of an optimal SFA for the indexed terms of Figure 43a begins with the computation of its cost, using Equation 12.1. The root position (*Pos*) and cost (*Cost*) of the lowest cost sub-SFA computed for a subsequence with end points (i,i') at any point in the computation are stored in a table (see Figure 43b) at the entry (i,i'), where i is the row and i' the column. Boldface numbers in the table represent the position and cost for the optimal subautomaton computed for the corresponding subsequence, while italic numbers represent the position and the cost of the discarded (sub-optimal) sub-SFA.

```
p(a,a,a).
p(b,a,b).
p(c,a,b).
p(c,b,b).
p(d,c,b).
p(d,c,c).
```

(a)

i \ i'	1	2	3	4	5	6
1	*Pos* *Cost* ***** **0**		*Pos* *Cost* *1* *6* **3** **5**			*Pos* *Cost* *1* *14* **2** **13** *3* *14*
2		*Pos* *Cost* ***** **0**	*Pos* *Cost* **1** **2**		*Pos* *Cost* **1** **7** *2* *7*	
3			*Pos* *Cost* ***** **0**	*Pos* *Cost* **2** **2**		
4				*Pos* *Cost* ***** **0**		
5					*Pos* *Cost* ***** **0**	*Pos* *Cost* **3** **2**
6						*Pos* *Cost* ***** **0**

(b)



(c)

Figure 43: Optimal SFA construction: predicate (a), cost table (b), SFA (c)

13. Multiterm indexing

In this section we deal with multiterm indexing. There are two cases when the need in multiterm indexing arises:

- The retrieval condition is specified in terms of a finite set of query terms rather than a single query term.
- We deal with an indexed set of *clauses* rather than single terms.

Typical retrieval conditions for multiterm indexing are:

- simultaneous unification;
- forward subsumption;
- backward subsumption;
- clause variance [Riazanov and Voronkov 2001*b*].

We introduce several definitions and then discuss these retrieval conditions in detail.

13.1. DEFINITION *(clause).* A *clause* is usually defined to be a finite set (or multiset) of literals. For the purposes of this section, it is enough to consider a clause as a *set of terms*. A clause which is a singleton set is called a *unit clause*. When we deal with clauses that may have cardinality ≥ 2 we will speak about *multiliteral clauses*.

We will denote clauses either as a sequence of their terms, $t_1, \ldots, t_n$, or as a disjunction $t_1 \vee \ldots \vee t_n$.

13.1. Simultaneous unification

Simultaneous unification is the following retrieval condition. Given a set of indexed terms $\mathcal{L}$ and a *sequence* of query terms $t_1, \ldots, t_n$, find all sequences of indexed terms $l_1, \ldots, l_n$ such that there exist substitutions $\sigma_1, \ldots, \sigma_n, \sigma$ such that

$$l_1\sigma_1 = t_1\sigma, \ldots, l_n\sigma_n = t_n\sigma.$$

Simultaneous unification is useful for implementing the hyperresolution and unit-resulting resolution inference rules. The hyperresolution inference rule can be formulated as follows:

$$\frac{l_1 \vee C_1 \quad \cdots \quad l_n \vee C_n \qquad \neg t_1 \vee \ldots \vee \neg t_n \vee D}{(C_1 \vee \ldots \vee C_n \vee D)\sigma},$$

where σ is a most general unifier of the sequences $(l_1, \ldots, l_n)$ and $(t_1, \ldots, t_n)$. In this rule, the premises are assumed to be variable-disjoint. However in practice, $t_i \vee C_i$ and $t_j \vee C_j$ for $i \neq j$ can be two copies of the same clause, so l_i and l_j will be the same term in the index. So we use $\sigma_1, \ldots, \sigma_n$ instead of σ in the corresponding retrieval condition.

A hyperresolution inference can be considered as a sequence of binary resolution inferences, but implementation of hyperresolution through binary resolution can be expensive. Hyperresolution is especially valuable when the input set of clauses consists of Horn clauses only. Then the rule can be reformulated in a simpler way:

$$\frac{l_1 \quad \cdots \quad l_n \qquad \neg t_1 \vee \ldots \vee \neg t_n \vee t}{t\sigma},$$

so only unit clauses have to be dealt with.

13.2. Subsumption

Multiliteral subsumption is the following retrieval condition on clauses. Given two clauses C and D, does there exist a substitution τ such that $C\tau \subseteq D$. If such a substitution exists, we say that C *subsumes* D. If both C and D are singletons, i.e., $C = \{s\}$ and $D = \{t\}$, it is easy to see that C subsumes D if and only if t is an instance of s. Therefore, when we retrieve candidates for subsumption among unit clauses, we can use the indexing techniques developed so far, retrieval of generalizations for forward subsumption and retrieval of instances for backward subsumption.

Among the retrieval conditions considered so far we distinguished two instance-related problems: retrieval of instances and retrieval of generalizations. Their analogues in multiterm indexing are respectively backward and forward subsumption.

13.2. Definition. *Backward subsumption* is the following retrieval condition: given a set of indexed clauses $\mathcal{C}$ and a query clause D, find all clauses $C \in \mathcal{C}$ such that D subsumes C. *Forward subsumption* is the following retrieval condition: given a set of indexed clauses $\mathcal{C}$ and a query clause D, does there exist a clause $C \in \mathcal{C}$ such that C subsumes D.

Subsumption on unit clauses reduces to matching, so it can be checked in linear time. Multiliteral subsumption is NP-complete. If we consider the set of indexed clauses as constant and vary only the query clause, then forward subsumption can be solved in polynomial time while backward subsumption is still NP-complete. Some provers treat multiliteral clauses as multisets of literals. Then a clause C subsumes a clause D if there exists a substitution σ such that $C\sigma$ is a *submultiset* of D. On multisets, subsumption is still NP-complete, but both forward and backward subsumption are polynomial. Moreover, backward subsumption on multisets can be checked in the time constant in the size of the query clause, but the constant can be exponential in the size of the index.

In practice, both forward and backward subsumption are very expensive operations. For some benchmarks forward subsumption may take over 90% of the overall running time of a prover. Forward subsumption is indispensable: with forward subsumption turned off resolution-based provers are unable to solve problems that are rather trivially solved with forward subsumption turned on. Backward subsumption works very well on some problems but is not very useful on other problems. For example, on some problems Vampire forward subsumes several million clauses while no clause at all is backward subsumed. For this reason, some provers employ incomplete backward subsumption algorithms or even turn backward subsumption off on some problems. Some provers, for example E, use incomplete forward subsumption algorithms.

13.3. Algorithms and data structures for multiterm indexing

There is a variety of algorithms and data structures for multiterm indexing. The techniques used in multiterm indexing are based on the techniques for term indexing. However, generalization to the multiterm case can be quite nontrivial, especially when one wants to achieve perfect filtering.

There are two main approaches to multiterm indexing.

1. Use the standard term indexing data structures and index on a small subset of the set of query terms, for example on two literals only. In this case the candidate set is obtained by the intersection of the candidate sets for the selected query terms. The size of the candidate set may be very large compared to the set of all compatible indexed clauses.
2. Design special data structures, obtained by modification of the corresponding data structures for term indexing. The retrieval algorithms may be quite complex compared to their term indexing counterparts. The aim is to achieve perfect filtering.

There are some issues to multiterm indexing which do not occur in term indexing. Typical examples are the following.

1. *The order of terms in the query set matters.* Consider the situation when the query set consists of two terms t_1, t_2 such that the set of candidate clauses for t_2 is empty while the set of candidate terms for t_1 is large. Obviously, it is better to begin retrieval with t_2 than with t_1, then the retrieval of candidates for t_2 can be omitted. Moreover, adaptive retrieval algorithms may play a major role.
2. For imperfect filtering based on selection of a subset of query terms *selection of the terms matters.* Ideally, the terms which give smaller candidate sets should be preferred.
3. For some indexing techniques, the *indexed clauses may be reordered before insertion into index*, to achieve a better degree of sharing and/or faster retrieval.

We will illustrate some of these concepts, when we consider code trees for forward subsumption.

13.4. Algorithms for multiliteral forward subsumption

A subsumption algorithm for multiliteral forward subsumption can be obtained from a subsumption algorithm for forward subsumption on terms in the following way. Suppose that we have an indexed clause $C = l_1, \ldots, l_n$ and a query clause $D = t_1, \ldots, t_m$. The clause C subsumes D if for every term l_i in C there exist a term

t_{k_i} in D such that l_i subsumes t_{k_i} with some substitution θ_i, and the substitutions $\theta_1, \ldots, \theta_n$ agree on all variables on which they are defined.

For example, let $C = \{g(x_1, f(x_2)), g(x_1, f(f(x_3)))\}$ and $D = \{g(y, f(f(y)))\}$. Then the first literal of C subsumes the first literal of D with the substitution $\theta_1 = \{x_1 \mapsto y, x_2 \mapsto f(y)\}$, and the second literal of C subsumes the first literal of D with the substitution $\theta_2 = \{x_1 \mapsto y, x_3 \mapsto y\}$. The substitutions θ_1 and θ_2 are both defined on x_1, and $\theta_1(x_1) = \theta_2(x_1)$, so C subsumes D.

The main difference between the subsumption algorithm for multiliteral clauses and that for terms is that for each term l_i of the indexed clause $C = l_1, \ldots, l_n$ we have to *find* a corresponding term t_{k_i} of the query clause D. The search for t_{k_i} can be implemented, for example, by a backtracking algorithm which tries the values $1, \ldots, m$ for k_i one by one.

To define a multiliteral subsumption algorithm, let us define some notions related to positions in a clause. From now on we view a clause as a *sequence* of its literals rather than a set.

Consider a clause $C = l_1, \ldots, l_n$. A *position in* C is any pair (m, p) such that $1 \leq m \leq n$ and each p is a proper position in l_m. The order $<$ on positions in C is defined as follows: $(m, p) < (m', p')$ if either $m < m'$ or ($m = m'$ and $p < p'$). As in the case of terms, we extend the set of positions in C by a special position ε called the *end position* in C; this positions is the greatest in the ordering $<$.

We will now define the functions next_C and after_C similar to the corresponding functions on terms. Let $\Lambda = p_1 < \ldots < p_m < p_{m+1} = \varepsilon$ be all positions in C. Then $\mathit{next}_C(p_i) = p_{i+1}$ for all $i \leq m$. The definition of after_C is as follows:

$$\mathit{after}_C(m, p) = \begin{cases} (m, \mathit{after}_{l_m}(p)), & \text{if } \mathit{after}_{l_m}(p) \neq \varepsilon; \\ \mathit{undefined}, & \text{otherwise} \end{cases}$$

We introduce a new relation and two functions on positions in clauses which are related to the literal structure. Let $C = l_1, \ldots, l_n$ be a clause whose proper positions are $p_1, \ldots, p_m$. The partial functions $\mathit{next\_literal}_C$ and $\mathit{previous\_literal}_C$ are defined on the positions of C of the form (m, Λ) as follows:

$$\mathit{next\_literal}_C(m, \Lambda) \quad = \quad \begin{cases} (m+1, \Lambda), & \text{if } m < n; \\ \varepsilon, & \text{otherwise.} \end{cases}$$

$$\mathit{previous\_literal}_C(m, \Lambda) \quad = \quad \begin{cases} (m-1, \Lambda), & \text{if } m > 1; \\ \varepsilon, & \text{otherwise.} \end{cases}$$

One can introduce a data structure on which the functions next_C, after_C, $\mathit{next\_literal}_C$, $\mathit{previous\_literal}_C$ can be evaluated efficiently. For example, one can use a double-linked list of flatterms.

The notions of *normalized form* of a clause C, denoted by $\mathit{norm}(C)$ and *variable equivalence relation* for a clause C, denoted $\mathcal{E}_C$, are defined in the same way as for terms.

Now we can define a clause-to-clause subsumption algorithm. Such an algorithm is given in Figure 44. Some parts of this algorithm are similar to the corresponding parts of the term subsumption algorithm, other parts implement iteration of literals $l_1, \ldots, l_n$ in the indexed clause C, search for t_{k_i} in the clause D, and backtracking. We use several labels and goto statements because the resulting algorithm will be easier to transform into instructions of an abstract subsumption machine than a more structured algorithm.

The algorithm traverses literals in C one by one (see the **first** label). If a match for a particular literal l_i was not found, the algorithm tries to use a different match for the previous literal l_{i-1} (see the use of $\mathit{previous\_literal}_C$ in **backtrack**). If a match of a particular literal in D against l_i fails, the next literal in D is tried (see the use of $\mathit{next\_literal}_D$ in **backtrack**). To remember the latest tried literal in D, we use the stack *backtrPos* of literal positions in D. When a new literal in D is tried, the position of this literal is pushed on *backtrPos*. This position is popped from *backtrPos* upon backtracking.

13.5. Code trees for forward subsumption

As in the case of term subsumption, we will now specialize the general subsumption algorithm *Subsume* for each indexed term C, obtaining its specialized version $\mathit{Subsume}_C$, and then represented the specialized version as a sequence of instructions of an abstract machine. The specialized version has the property $\mathit{Subsume}_C(D) = \mathit{Subsume}(C, D)$, for each query term D. The instructions of the abstract machine are very similar to those use for term subsumption, except that added are new instructions for iterating literals in the indexed and query clauses and for backtracking. The code for each particular literal in the indexed clause is compiled almost in the same way as for the term subsumption, except that instead of the label *faill* we use the label of the appropriate backtracking instruction.

13.3. Example. Let $C = g(x_1, c), h(x_1, x_1)$ be an indexed clause. The specialised version of the subsumption algorithm for this clause and the corresponding sequence of instructions are shown in Figures 45 and 46.

In order to represent the specialized algorithm, two new kinds of instructions have been added: *First* and *Backtrack*. The semantics of these instructions in code trees is given in Figure 47. The instruction *First* sets the current position p to the first position in D and memorizes the address of the following *Backtrack* instruction in the backtrack stack *backtrInst*. The instruction *Backtrack* iterates over other literals in D. This instruction can be reexecuted several times, since it repeatedly pushes its own address on the backtrack stack, until no more literals in the query clause can be found.

Codes for several clauses can be compiled into a code tree, as in the case of forward subsumption for unit clauses. However, we obtain a more considerable gain in efficiency for multiliteral clauses when the backtrack points in different clauses

```
procedure Subsume(C, D)
begin
  let subst be an array for storing positions in C;
  let backtrPos be a stack for storing positions in D;
  pos_C := (0, Λ);
  first: /* try the first literal in D */
    pos_C := next_literal_C(pos_C);
    pos_D := (1, Λ);
    push(Λ, backtrPos);
    goto next;
  backtrack:
    pos_D := next_literal_D(pop(backtrPos));
    if pos_D = ε then
      if pos_C = Λ then return failure;
      pos_C := previous_literal_C(pos_C);
      goto backtrack;
    else
      push(pos_D, backtrPos);
    endif
  next: /* trying the next literal in D */
    do
      if norm(C)/pos_C = *_i then
        subst[i] := pos_D;
        pos_D := after_D(pos_D);
        pos_C := after_C(pos_C);
      else /* C/pos_C is not a variable */
        if root(C/pos_C) ≠ root(D/pos_D) then goto backtrack;
        pos_D := next_D(pos_D);
        pos_C := next_C(pos_C);
      endif;
    while pos_C is defined;
  /* literal matched successfully */
  if pos_C ≠ ε goto first;
  /* All literals traversed, comparison of terms */
  let ⟨u_1, v_1⟩, ..., ⟨u_n, v_n⟩ be the canonical computation sequence for ℰ_C.
  forall i ∈ {1, ..., n}
    if D/subst[u_i] ≠ D/subst[v_i] then goto backtrack;
  return success;
end
```

Figure 44: A clause-to-clause subsumption algorithm

```
procedure Subsume_C(D)
begin
  first1:
    p := Λ;
    push(Λ, backtrPos);
    goto next1;
  backtrack1:
    p := next_literal_D(pop(backtrPos));
    if p = ε then return failure;
    push(p, backtrPos);
  next1:
    if root(D/p) ≠ g then goto backtrack1;
    p := next_D(p);
    subst[1] := p;
    p := after_D(p);
    if root(D/p) ≠ c then goto backtrack1;
    p := next_D(p);
  first2:
    p := Λ;
    push(Λ, backtrPos);
    goto next2;
  backtrack2:
    p := next_literal_D(pop(backtrPos));
    if p = ε then goto backtrack1;
    push(p, backtrPos);
  next2:
    if root(D/p) ≠ h then goto backtrack2;
    p := next_D(p);
    subst[2] := p;
    p := after_D(p);
    subst[3] := p;
    p := after_D(p);
  if D/subst[1] ≠ D/subst[2]
    then goto backtrack2;
  if D/subst[1] ≠ D/subst[3]
    then goto backtrack2;
  return success;
end
```

Figure 45: The algorithm *Subsume* specialized for the clause $g(x_1, c), h(x_1, x_1)$

```
initl : Initialize(m_1)
m_1  : First(m_3, m_2)
m_2  : Backtrack(m_3)
m_3  : Check(g, m_4, faill)
m_4  : Put(1, m_5, faill)
m_5  : Check(c, m_6, faill)
m_6  : First(m_8, m_7)
m_7  : Backtrack(m_8)
m_8  : Check(h, m_9, faill)
m_9  : Put(2, m_10, faill)
m_10 : Put(3, m_11, faill)
m_11 : Compare(1, 2, m_12, faill)
m_12 : Compare(1, 3, m_13, faill)
m_13 : Success
faill : Failure
```

Figure 46: The corresponding sequence of instructions

$First(m_1, m_2)$	$p := \Lambda$; $push(m_2, backtrInst)$; $push(\Lambda, backtrPos)$; goto m_1;
$Backtrack(m_1)$	$p := next\_literal_D(pop(backtrPos))$; if $p = \varepsilon$ then goto $faill$; $push(self, backtrInst)$; $push(p, backtrPos)$; goto m_1;

Figure 47: Semantics of additional instructions in code trees

are shared. This happens when the normal forms of one or more initial literals in the index clauses coincide. Consider, for example, two clauses $g(x_1, c), h(x_1, x_1)$ and $g(x_1, c), h(c, x_1)$. The code tree for these clauses and the dataflow of the retrieval algorithm are shown in Figure 48. The very expensive backtracking-related instructions are shared, both for the first and for the second literal in the clauses.

14. Issues in perfect filtering

Up to now, we did not pay major attention to perfect filtering. For some applications, perfect filtering is easy to achieve. A typical example is search for generalizations when all indexed terms are linear (functional programming).

In theorem proving perfect filtering is important because the sets of indexed terms often easily contain 10^5–10^6 terms, so imperfect filtering can results in too many candidates or too few candidates. In this section we consider issues that arise in perfect filtering.

14.1. Normalization of variables

The most typical source of imperfect filters is normalization of terms which "forgets" the variable names. For example, if the term $f(x, x)$ is stored in the index as $f(*_1, *_2)$, one cannot achieve perfect filtering for any of standard retrieval conditions. There are two ways of coping with this problem: normalization of variables and equality constraints.

Normalization of variables. Variables are enumerated in the order of their first occurrences in the indexed term. For example, both terms $f(x, y, x)$ and $f(y, x, y)$ will be turned into $f(*_1, *_2, *_1)$. The main disadvantage of this technique is

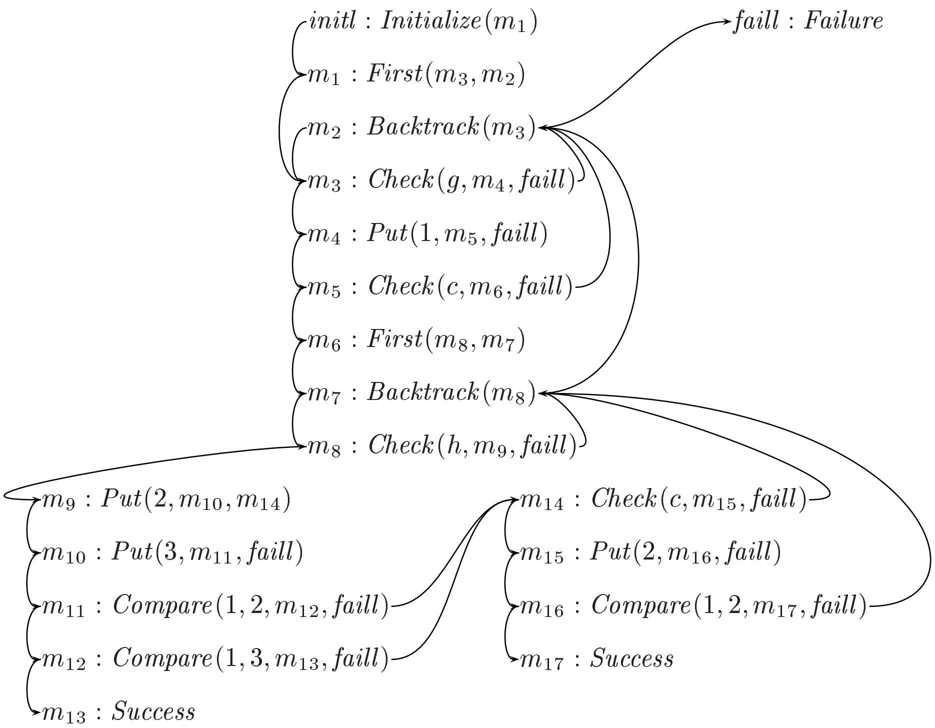


Figure 48: Code tree for two clauses $g(x_1, c), h(x_1, x_1)$ and $g(x_1, c), h(c, x_1)$

a relatively small degree of sharing, when indexed terms contain many occurrences of variables. To illustrate this, consider Figure 49 showing three techniques of representing variables in an index storing the set of two terms $\{a(x_1, x_1, e(x_2, x_2)),\ a(x_1, x_2, e(x_1, x_1))\}$ in a discrimination tree.

In (a) imperfect filtering is used, in (b) variables are enumerated, and in (c) equality constraints (described below) are used. Of the three techniques, variable enumeration is likely to consume the largest amount of space when the term sets are large.

Equality constraints. An indexed term l is considered as a pair $\langle l', E\rangle$, where l' is obtained from l by "forgetting" variable names as in the case of imperfect filtering; and E is a set of *equality constraints* of the form $*_i = *_j$ which encode dependencies among the variables of l. A pair $*_i = *_j$ belongs to the equality constraint if $i < j$ and the variable $*_i$ represents the first occurrence of $*_j$ in l. For example $f(x_1, x_1, g(x_2, x_1))$ will be represented by the pair $\langle f(*_1, *_2, g(*_3, *_4)), \{*_1 = *_2, *_1 = *_4\}\rangle$. The index consists of nodes of two kinds:

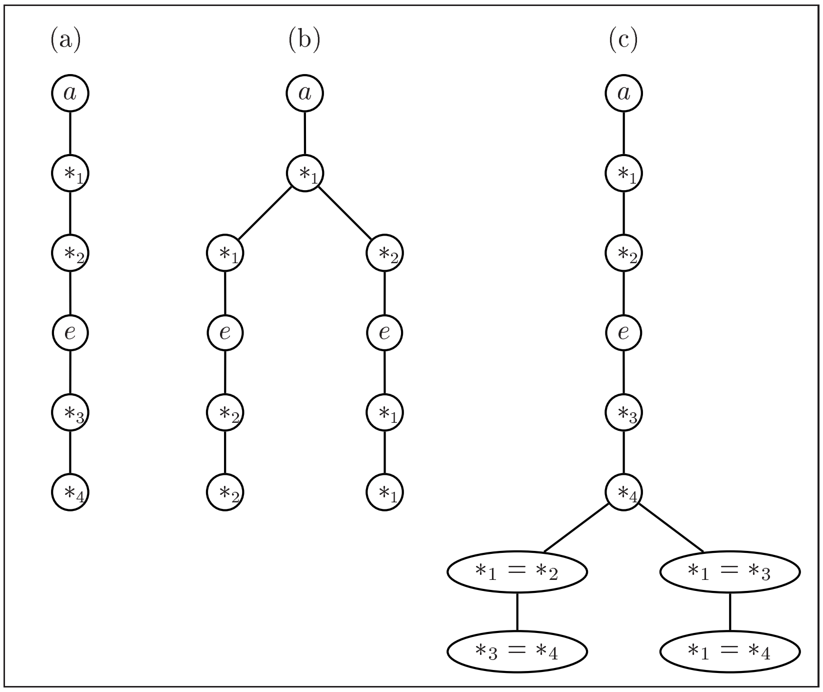


Figure 49: Three ways of representing variables: (a) using imperfect filtering; (b) enumerating variables; (c) using equality constraints

ordinary nodes plus equality constraints.

Using equality constraints results in indexes of smaller sizes. Usually, their use reduces both time for retrieval and memory used by the index. Retrieval is usually faster due to the following observation: each variable creates a potential backtrack point. When variable enumeration is used, each different variable creates an "early" backtrack point. When equality constraints are used, these backtrack points are delayed until the whole term has been processed.

When perfect filtering is required, another problem may arise, both for variable enumeration and equality constraint techniques. This problem arises when multiliteral clauses are used. In multiliteral clauses, one cannot normalize variables in a single literal, because variables are not local to a literal, but may be shared by different literals in a clause. The only sound way to normalize variables is by normalizing them within a clause. Thus, a clause $P(x, y), Q(y, z)$ will be normalized into $P(*_1, *_2), Q(*_2, *_3)$. If we want to store both terms $P(*_1, *_2)$ and $Q(*_2, *_3)$ in the index, then the index may contain nonnormalized literals, for example, $Q(*_2, *_3)$ instead of $Q(*_1, *_2)$. Storing terms that are not normalize can decrease sharing in the index. An interesting technique implemented in some indexes in Vampire is storing normalized terms plus substitutions. For example, when we have to store

$Q(*_2, *_3)$, we will store $Q(*_1, *_2)$ instead, but store it together with the substitution $\{*_1 \mapsto *_2, *_2 \mapsto *_3\}$.

14.2. Multiple copies of variables

There is a problem that arises in some indexing operations: for some indexed terms, their multiple copies should be used. A typical example is when search for generalizations is used to reduce a term to its normal form. During reduction to normal form, a term can be rewritten several times, by the same indexed terms, but with different substitutions. For example, if we derived an ordered unit equality $f(x) = g(x)$, it can be used twice to rewrite the query term $h(f(f(y))$. The first rewriting uses the substitution $\{x \mapsto f(y)\}$ to rewrite the query term into $h(g(f(y))$; then the second rewriting uses the substitution $\{x \mapsto y\}$ to rewrite this term into $h(g(g(y))$. When such a sequence of rewritings is performed, several different substitutions can be made for the variables of an indexed term. Therefore, representation of variables (or substitutions) in the index must allow for several different substitutions to be done.

Several provers, including Otter, Vampire, and Fiesta, use *variable banks* use discussed above in Section 3.2. A variable bank is a data structure (usually an array) which represents a (substitution for a) collection of variables. For example, if the index uses variables $*_1, \ldots, *_n$, we can use an array of size n whose elements are references to terms. Each time a sequence of indexing operation is to be performed, for each operation in the index a new variable bank is used.

15. Indexing modulo AC-theories

Many mathematical functions used and modelled in automated reasoning are associative and commutative, as captured by the following two axioms:

$$f(x, f(y, z)) = f(f(x, y), z); \qquad (A)$$
$$f(x, y) = f(y, x). \qquad (C)$$

We write $f \in AC$ if f is such an associative-commutative symbol and write $s =_{AC} t$ to indicate that s and t are equivalent under associativity and commutativity.

15.1. Definition *(Flattened Term). Flattened representation* of a term t is obtained by reducing it to its normal form using rewrite rules of the following form for each AC-symbol f:

$$f(X, f(Y), Z) \quad \to \quad f(X, Y, Z).$$

Here X, Y and Z denote arbitrary sequences of terms. We denote the flattened form of a term t by $\overline{t}$.

In this section we will often use the flatten form of a term instead of the term itself, and thus treat $f \in AC$ as a function symbol of a nonfixed arity.

AC-indexing refers to the variation of indexing problem that arises when the relationship between the indexed and query term is augmented to take the AC-properties into account. In this section, we discuss an indexing technique for selecting AC-generalizations of a given query term. More formally, the problem is to select those indexed terms l such that there exists a substitution σ with $l\sigma =_{AC} t$. If such a substitution exists we say that t *AC-matches* l, l is an *AC-generalization* of t, and t is an *AC-instance* of l.

Consider an example. Let $f \in AC$, $l = f(a,x,b)$, and $t = f(b,c,d,a)$. Then for $\sigma = \{x \mapsto f(c,d)\}$ we have $l\sigma =_{AC} t$. AC-matching has been studied extensively (see, for example [Gottlob and Leitsch 1987, Benanav, Kapur and Narendran 1987, Socher 1989, Kounalis and Lugiez 1991, Nicolaita 1992, Verma and Ramakrishnan 1992, Lugiez and Moysset 1993, Bachmair, Chen and Ramakrishnan 1993, Bachmair, Chen, Ramakrishnan, Anantharaman and Chabin 1995, Eker 1995]). AC-matching is NP-complete in general but can be performed in polynomial time for linear query terms [Benanav et al. 1987].

If $f \in AC$, we use the notation $\sim$ to denote the smallest symmetric rewrite relation (also called the *permutation congruence*) for which $f(X,u,Y,v,Z) \sim f(X,v,Y,u,Z)$. Observe that l is an AC-generalization of t if and only if $\bar{l}\sigma \sim \bar{t}$, for some substitution σ.

The *top-layer* of a term l, denoted by $\hat{l}$, is obtained by replacing all occurrences of an AC-symbol f by a constant, also denoted by f. For example, if $f \in AC$ and $g \notin AC$, then the top-layer of $g(x,f(b,f(x,c)))$ is $g(x,f)$. Let l and t be two flattened terms with $l\sigma \sim t$. Then we have

(i) $\hat{l} \leq \hat{t}$. For example, the term $g(a,g(a,a))$ cannot be an AC-instance of $g(x,f(b,c)$ as the respective top-layers do not match.

(ii) if p is a position of an AC-symbol in the top-layer of l, then $(l/p)\sigma \sim t/p$, i.e., if the top-layers of the given terms match,then we need to consider the stripped-off subterms at the positions of f and recursively determine whether suitable AC-matches can be found.

Conversely, if (i) and (ii) are satisfied for some substitution σ, then $l\sigma \sim t$. In other words, AC-matching is completely characterized by conditions (i) and (ii).

Condition (i) represents a standard (i.e., non-AC) matching problem. Condition (ii) leads to a bipartite graph matching problem as shown below. Suppose that $l/p = f(l_1,\ldots,l_m)$ and $t/p = f(t_1,\ldots,t_n)$, where $f \in AC$. Let us also assume, without loss of generality, that for some k, $0 \leq k \leq m$, no term $l_1,\ldots,l_k$ is a variable, while all terms $l_{k+1},\ldots,l_m$ are variables. Define a bipartite graph $G = (V_1 \cup V_2, E)$, with $V_1 = \{t_1,\ldots,t_n\}$, $V_2 = \{l_1,\ldots,l_k\}$, and E consisting of all pairs (t_i,l_j), such that $l_j\sigma \sim t_i$, for some substitution σ. It can easily be seen that if

- $n = m$ or $n > m > k$, and
- there is a maximum bipartite matching of size k in the bipartite graph G

then $f(l_1,\ldots,l_m)\sigma \sim f(t_1,\ldots,t_n)$, for some substitution σ and hence condition (ii)

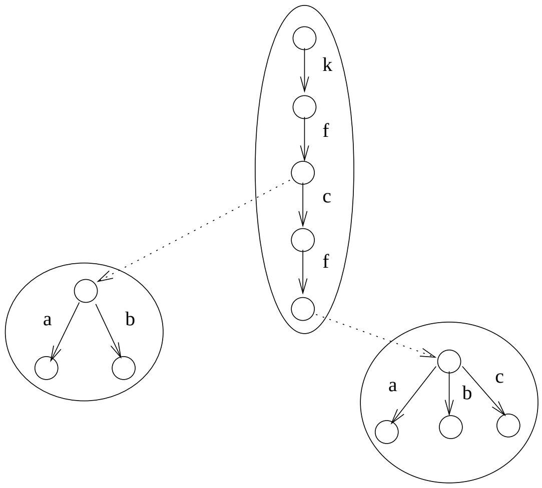


Figure 50: Example of an AC-discrimination tree.

is satisfied.

Based on the above discussion, we can perform indexing in two phases. In the first phase, we use *associative-commutative (or AC-) discrimination trees* for indexing. An *AC-discrimination tree* for a set of terms $\mathcal{I}$ is a hierarchically structured collection of standard discrimination trees. The top of the hierarchy is a standard discrimination tree for the set of top-layers of terms in $\mathcal{I}$; and at each node with an incoming edge labeled by an AC-symbol another AC-discrimination tree is attached that represents the corresponding set of nonvariable subterms of terms in $\mathcal{I}$.

The AC-discrimination tree for the terms

$$k(f(a,b),c,f(y,c)) \text{ and } k(f(a,x),c,f(a,b))$$

is shown in Figure 50. The associated top-layers are $k(f,c,f)$ and $k(f,c,f)$. The first subtree represents the terms $f(a,b)$ and $f(a,x)$, the second subtree the terms $f(y,c)$ and $f(a,b)$.

To use this tree, we

- traverse the individual standard trees in the hierarchy as usual, and then
- use bipartite graph matching to combine the results from different levels of the hierarchy.

To illustrate this process, consider using the above tree to retrieve generalizations of $k(f(b,a),c,f(a,b))$. The top-level tree is successfully traversed to the first AC-node, call it v, at which point the two subterms b and a are provided as inputs to the first subtree. Traversal of the subtree yields two bipartite graphs, one for each element of the set $L_v = \{f(a,b), f(a,x)\}$. The size of each respective maximum matching indicates that the input term $f(b,a)$ is an AC-instance of both $f(a,b)$ and $f(a,x)$. Traversal of the top-level tree resumes at node v and continues on to the second AC-node, call it v', with $L\text{-}v' = \{f(y,c), f(a,b)\}$. The input terms for the second subtree are a and b. The subterm $f(a,b)$ of the input is found to be an AC-instance of $f(a,b)$, but not $f(y,c)$. Thus, only the indexed term $k(f(a,x),c,f(a,b))$ is identified as an AC-generalization. The running time for the first stage (i.e., AC-matching linear terms) using AC-Discrimination trees has polynomial complexity. In fact, the running time of the algorithm is dominated by the cost of doing bipartite graph matching, which takes $O(mn^{1.5})$, where n is the size of t and m is the sum of the sizes of all indexed terms in $\mathcal{I}$.

The bipartite graphs that arise are of the form $G = (V_1 \cup V_2, E)$, where $V_1 = \{t_1, \ldots, t_n\}$ is a set of subterms of the query term, $V_2 = \{l_1, \ldots, l_k\}$ is a set of indexed subterms, and E contains an edge (t_i, l_j) if and only if l_j AC-matches t_i. Note that V_2 depends only on the indexed terms; it remains fixed for a given AC-tree and a node therein. We can design special techniques to efficiently handle bipartite graph matching for cases where V_2 is small, say $k \leq 4$. The graph G is computed stepwise, via a sequence of bipartite graphs $G_1, G_2, \ldots, G_n$. Traversal of the subtree for query t_1 determines the edges incident on that node, thereby defining G_1. Traversal of the subtree for t_2 determines the edges incident on t_2, which are added to G_1 to yield G_2; and so on.

Suppose $V_2 = \{t_1, t_2\}$. Each traversal of the subtree for a term s_i yields a bitstring b_i of length two; the bitstring 10 is obtained if s_i is an AC-instance of t_1 but not of t_2; and the strings 00, 01 and 11 are interpreted correspondingly. The size of a maximum matching on the bipartite graph G_i can be readily determined from b_i and the size of a maximum matching on G_{i-1}. We compute this efficiently using *Bipartite Graph Matching Automata.* [9] Let $|V_2| = k$ and Π_d denote all subsets of d vertices ($d \leq k$). Encode by a state all possible bipartite graphs that have maximum bipartite matchings of size d and incident on one or more of the subsets in Π_d. Figure 51 shows an example bipartite graph matching automata for the case $d = 1, k = 2$ and $\Pi_1 = \{\{1\}, \{2\}\}$.

We use the adjacency matrix to represent the bipartite graph. The element in the ith row and jth column is 1 if and only if s_i AC-matches t_j. The symbol on which a transition is made is a bitstring of length k (each such bitstring represents a possible row in the adjacency matrix). A transition is made from a state representing a maximum matching of size d to a state representing a maximum matchings of size $d+1$. Denote by S_k the bipartite graph matching automata for finding maximum matchings of size k. For $|V_2| = k$ use S_k.

The initial state of the automaton represents graphs with a maximum matching of size 0. Its three successor states represents three different types of graphs with maximum matchings of size 1. The three cases are: (i) t_1 can be matched, but not t_2; (ii) t_2 can be matched, but not t_1; (iii) both t_1 and t_2 can be matched, but not at the same time. The final state represents graphs with a maximum matching of size 2. Finally, we read the rows of the adjacency matrix representing the bipartite

[9] Bachmair et al. [1993] call these *Secondary Automata.*

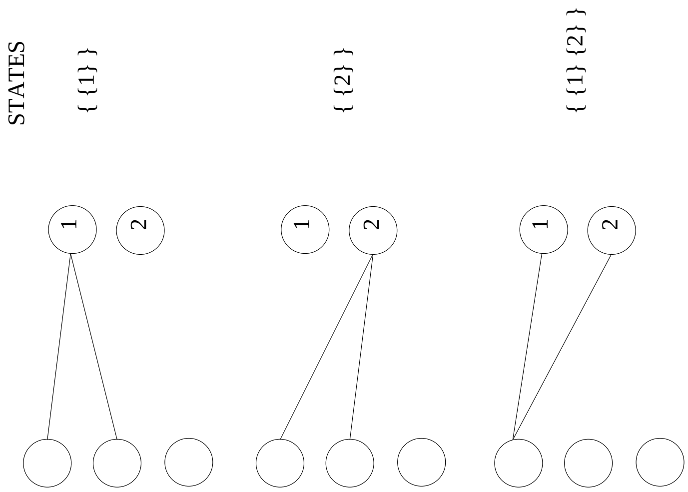


Figure 51: Example of bipartite graph matching automata.

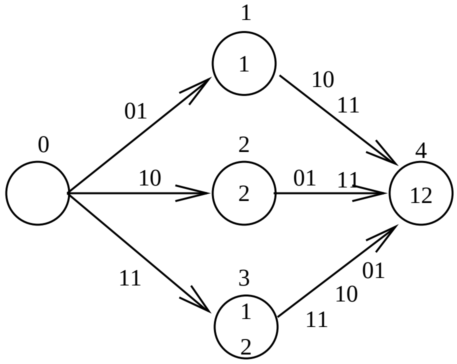


Figure 52: Example

graph one by one and make transitions accordingly. There is a maximum bipartite matching of size k if and only if the final state in S_k is reached.

By using bipartite graph matching automata in conjunction with AC-discrimination trees for doing AC-matching we can show that for the first stage, all bipartite matching problems at all AC-nodes visited during traversal of an AC-discrimination tree for a set of flat terms with total size m, on an input term of size n can be done in $O(mn)$ time.

16. Elements of term indexing

In this section, we provide a list of "basic elements" of term indexing techniques such that new indexing techniques can be devised via a suitable combination of these elements. This enables us to understand many known indexing methods as instances of the generic framework. The common framework also makes it easier to compare and contrast the known approaches. Most importantly, the framework distills out essential characteristics of different techniques, so that (a) one can easily combine these characteristics to get new indexing techniques, (b) understand the trade-offs involved and be able to predict/explain performance of different techniques.

16.1. Deterministic and backtracking automata

An indexing automaton that requires reinspection of symbols in the query term is called a *backtracking automaton*. If all query terms can be processed without such reinspection, the automaton is said to be deterministic. The reinspection of symbols in the query term is necessitated when there exist two p-string prefixes (i.e., prefixes of p-strings) S_1 and S_2 such that:

- there exists a term t compatible with both S_1 and S_2;
- S_1 and S_2 are respectively of the form SS_1' and SS_2', where S is the maximal common prefix of S_1 and S_2;
- S_1' and S_2' examine at least one common position, say, p.

The first part of the condition implies that, when the index is traversed to find p-strings compatible with t, the states s_1 and s_2 representing S_1 and S_2 would have to be visited. Otherwise we would miss out some p-strings compatible with t. The second condition implies that neither of these states would be a descendant of another, but have a common ancestor s that represents their maximal common prefix S. The retrieval algorithm would thus follow the path from the root of the index to s and on to s_1, and then will need to backtrack back to s and then follow down the path to s_2. In this process, the symbol at p is first inspected on the path from s to s_1, and then inspected again on the path from s to s_2.

Techniques for avoiding reinspection fall into several different categories that can be related to the above criteria:

- use of augmented sets of indexed terms that enable one to ignore one of S_1 or S_2 completely, without losing correctness. In particular, this would require that every indexed term that is compatible with t can be found by following just the path to s_1; there would be no need to backtrack and retry the match by following the path to s_2.
- use of traversal orders that avoid generation of p-strings that satisfy the above criteria. For instance, path-indexing ensures that after visiting a common prefix S, two different path-strings would visit the same exact set of positions (if they correspond to the same root-to-leaf paths), or a completely disjoint set of positions (if they correspond to completely different paths).

- use of *failure links* that enable us to reuse the information seen about the symbol at p while matching S_2. The automata-driven indexing technique follows this approach, and thus avoids reexamination.

Of these, the first technique is the one that has been studied from the point of building deterministic indexes [Graf 1991, Sekar, Ramesh and Ramakrishnan 1995], so we consider only this technique in this section. Other techniques are discussed separately in the following sections.

To understand how a set of indexed terms can be augmented so as to avoid reinspection of symbols, consider the set of indexed terms

$$\{f(b, a, a), f(*, a, b), f(b, a, *)\}.$$

An index for retrieval of generalizations was illustrated in Figure 4(a). The p-strings $\langle \Lambda, f\rangle\langle 2, a\rangle\langle 3, b\rangle$ and $\langle \Lambda, f\rangle\langle 2, a\rangle\langle 3, *\rangle\langle 1, b\rangle$ satisfy the conditions laid out above that necessitate reinspection of symbols. These two strings correspond to S_1 and S_2 mentioned in the condition: both strings correspond to generalizations of the query term $f(b, a, b)$, with a common prefix $\langle \Lambda, f\rangle\langle 2, a\rangle$, and position 3 requiring reinspection. To avoid the reinspection, we can augment the above term set to include the terms $f(b, a, b)$ and $f(?, a, b)$. We then annotate each leaf of the index with the set of *all* indexed terms that generalizations of the p-string reaching that leaf. The new index that incorporates these modifications is shown in Figure 53(a). Retrieval of generalizations using this automaton requires no backtracking.

The augmented set of indexed terms is typically not constructed explicitly. Instead, it is constructed implicitly by the index construction algorithm. For instance, in the *Build* algorithm of Section 7, we define the notion of a match set so as to ensure that it is sufficient to follow down a single path in the index.

The trade-off in using deterministic automata is one of space (and construction time) versus retrieval time. Observe that these considerations are very similar to that of deterministic versus nondeterministic finite-state automata.

16.1.1. Issues and trade-offs

- Deterministic indexes require no backtracking for retrieval. The final states can also store substitutions, in addition to the candidate terms. These factors lead to fast retrieval times.
- Deterministic indexes are expensive in terms of space and construction/modification time—they can be exponential in size and number of indexed terms for retrieval of generalizations, possibly worse for other retrieval conditions. (*Compare with the trade-offs between NFAs and DFAs.*)

16.2. Traversal orders: multiple vs single p-string

One of the main decisions to be taken when developing an indexing technique is the order in which indexed terms are traversed to generate p-strings. Some indexing techniques use traversal orders that generate a single p-string from each indexed

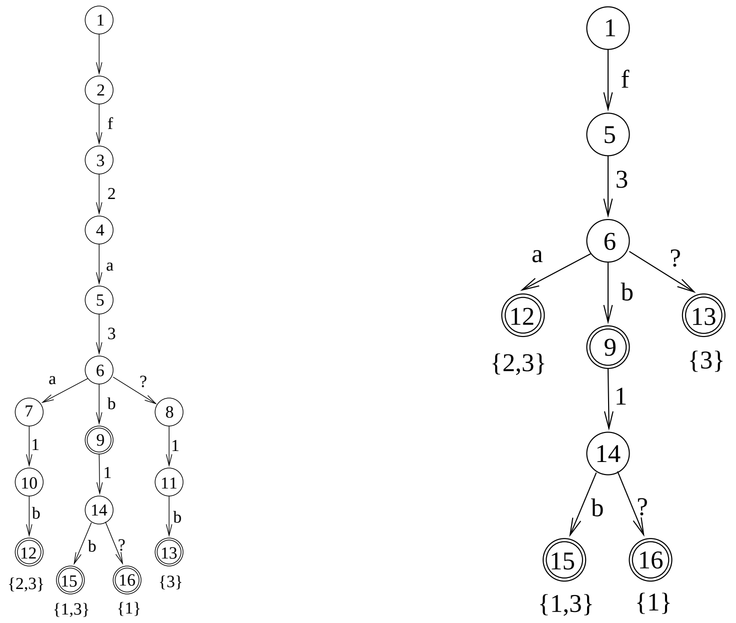


Figure 53: Deterministic and pruned/collapsed index

term, whereas others generate multiple p-strings. For instance, discrimination trees and adaptive automata generate single p-strings. In contrast, path indexing generates multiple p-strings from each indexed term. The primary advantage of generating a single p-string is that a compatible p-string directly yields a compatible indexed term. When multiple p-strings are generated from a single indexed term, we need to ensure compatibility of all these p-strings with the query term. Viewed in another way, nontrivial operations need to be performed on sets of compatible p-strings in order to obtain compatible indexed terms.

On the other hand, generating multiple strings typically leads to shorter strings, which in turn leads to increased sharing among strings, and thus reduced storage requirements. Moreover, techniques that generate multiple p-strings can avoid situations where reinspection of symbols may be needed. For instance, path indexing uses a traversal order where the conditions laid out in the previous section regarding symbol reinspection will never be satisfied.

An important consideration in the choice of traversal orders is whether they yield p-strings that contain variables (or '*' symbol in them). The presence of such embedded variables introduces the need for backtracking in an index. For instance, in the example discussed in the previous section, the need for reinspection of symbols arose because there were two p-strings such that the first position where they differ correspond to a variable in one of the p-strings, but a nonvariable in another. Similarly, scanning past a variable in the query term often requires us to have forward links for skipping appropriate subterms (e.g., jump-lists in discrimination trees).

16.2.1. Issues and trade-offs

- Path indexing produces many p-strings such that no variables are embedded within the strings. Moreover, these strings do not contain any pair of positions that do not have an ancestor-descendant relationship. The first property ensures that there is never a need to skip a subterm in a query term, while the second property ensures that we need not have jump lists that need to be taken when we inspect variables in the query term. The main results from the point of view of indexing are:
  - the automata are completely deterministic;
  - the automaton size (i.e., the number of nodes in it) is linear in the size of indexed terms;
  - but we require many operations for combining intermediate results (obtained as a result of matching individual p-strings) to compute the set of compatible terms.
- Automata-driven indexing produces fewer strings than paths, and shares the property with path indexing in that they do not contain embedded variable. Moreover, due to the nature of p-strings generated, there is no need to skip over subterms in the query term at runtime. The net result is that:
  - the automaton is completely deterministic;
  - it uses space quadratic in the size of the indexed terms;
  - this method also requires many operations for combining results of p-string matches to identify candidate terms, but the intermediate sets are always smaller than those for path indexing.
- Discrimination trees produce just a single p-string from each indexed term, which contains several embedded variables and subterms. The net effect is that:
  - indexing requires significant amount of backtracking (nondeterministic);
  - automata require significant storage for jump lists to skip subterms;
  - no effort needs to be spent for combining intermediate results.
- Adaptive automata produce p-strings with no embedded variables, but to do this, they generate many instances of the original set of indexed terms. The net effect for computing generalizations of a query term is that:
  - indexing requires no backtracking;
  - no need for combining intermediate results;
  - worst case space requirement is exponential in the size of indexed terms.

16.3. Adaptive and fixed-order traversals

For adaptive traversals, the next set of positions to inspect can depend on all of the symbols inspected so far. Fixed-order traversals, on the other hand, select the next position as a function of the fringe positions of the prefix inspected.

Adaptive traversal orders are utilized in adaptive automata, substitution trees, and unification factoring. Fixed-order traversals are used in path indexing, discrimination trees, and automata-driven indexing. The trade-offs in using adaptive versus fixed order of traversals is related to the complexity of algorithms for generating adaptive traversals (these algorithms can be simple or complex), as well as the interaction of adaptive traversal orders with other optimizations such as failure links.

16.3.1. Issues and trade-offs

- Adaptive traversals can minimize the number of variables embedded within p-strings, which reduces the amount of backtracking needed.
- Computing the traversal order may add overhead to index construction and maintenance operations, but retrieval time is improved.
- With fixed-order traversals, the only mechanism to minimize embedded variables is by producing multiple p-strings.

16.4. Pruned and collapsed indexes

We can borrow and/or adapt techniques for reducing space requirements for tries in order to reduce the space requirements for term matching. In particular, we can borrow the idea of *pruned tries*, where we stop examining symbols as soon as we identify a unique string that is compatible with the query string. We can adapt the idea of *collapsed tries*, where sequences of states all of which have a unique successor are eliminated from the trie. For both these techniques, the idea is that inspection of positions eliminated by the technique does not reduce our ability to discriminate among them.

We can apply both these techniques to term indexes as well. In particular, we can use the pruning technique to replace a subtree of the index that contains at most one final state by the final state. Note that such pruning does not adversely affect the accuracy of filtering — in the worst case, the query term may not be compatible with the p-string corresponding to the final state, but we have at most one indexed term that has been incorrectly identified as being compatible with the query term. This slight decrease in accuracy may be well worth the savings in space.

We can easily extend the above technique so that we not only replace subtrees with single final states, but also subtrees with less than k final states, for some small k. Once again, the decrease in accuracy (proportional to k) may be small as compared to the savings in space usage.

In order to apply the collapsing technique, note that we may not be able to eliminate some intermediate states because they examine positions whose children will later be examined. Nevertheless, we may be able to "compress" all such transitions using a single operation that examines multiple positions in the query term. We will use the term *augmented edge* to denote such edges.

Figure 53(b) shows the result of applying the pruning and collapsing techniques on the automaton of Figure 53. Note that the terminal sequence of states examining position 1 have been eliminated in the leftmost and rightmost paths in the indexing automaton. We have also eliminated the inspection of position 2, which occurred immediately after inspecting position 1 in Figure 53. Observe that we could eliminate this test since no children of a are being examined further down in the automaton. Had this not been the case, we would have collapsed the inspection of the root symbol and position 2. One possible way to represent this in the automaton is by using $f(*, a, *)$ on the outward transition from state 1, rather than the label f. Whether the collapsed representation leads to actual space reduction at the implementation level is unclear. If the index is represented as code as in Section 9, the collapsed representation may end up being as expensive as (or perhaps more expensive than) the uncollapsed representation. If the index is represented using traditional representations for finite-state machines, then collapsing can lead to some improvements in space usage. However, if significant amount of auxiliary information is to be stored with each trie state, then the storage savings due to collapsing can be significant.

Finally, note that we can apply the collapsing technique more generally to replace subcomponents of the trie that are deep but have few branches. We can collapse each root-to-leaf path (i.e., paths from the root of the subcomponent to its leaf, which may or may not coincide with the leaves of the whole trie) in such components into an augmented edge. We may use techniques such as hashing to identify which of the augmented edges are to be taken.

The benefits and trade-offs associated with pruned/collapsed indexes are as follows:

- Pruning and collapsing can save space, without necessarily losing discrimination (unlike techniques such as depth-bounding).
- Increased time for making transitions, as pruned edges are associated with complex conditions for transition. (Note that there would be fewer transitions that are taken, so the total time spent in making transitions may not be affected.)
- But overall retrieval time may be reduced, as these techniques enable incremental construction of good adaptive traversals — note that decision regarding order of inspection of symbols corresponding to pruned/collapsed portions of the trie is not fixed prematurely, but determined as more terms are inserted in the index.
- Note that each transition in the index may result in a binding operation to variables in the query term. By reducing the number of transitions (using pruning/collapsing) we can reduce the time spent in performing such bindings.

16.5. Failure links

Most of the indexing techniques studied so far have the property that, once a failure is encountered, they backtrack to the lowest ancestor node where either a '*' transition could have been taken, or a variable was encountered in the query term. At this point, they follow down the next possible transition from this state. If there is failure along this path, then we look for the next outermost branch and so on. In general, this can lead to wasted effort in rescanning portions of the query term again and again, as we follow down each of these alternate paths in the index. Failure links exploit information known about the structure of the query term at the point of failure in order to prune away alternate paths that are bound to fail.

Recall that failure links are another well-known concept in string matching, pioneered in the Knuth-Morris-Pratt algorithm [Knuth, Morris and Pratt 1977]. Among known indexing techniques, it has been used only in automata-driven indexing [Ramesh et al. 1990] and threaded automata for subsumption [Rao et al. 1996], although it is readily applicable to any backtracking index. One of the benefits of studying the indexing technique within a common framework is that we can easily identify components such as this which can be developed in the context of one technique and then applied to other techniques.

16.5.1. Issues and trade-offs

- reduces backtracking and/or the computation involved in backtracking;
- may increase space usage;
- increases the cost of index construction/maintenance operations.

16.6. Sharing equivalent states

For deterministic indexes, two states are equivalent if the subindexes rooted at these states are identical. By identifying such subautomata and merging them, we can build dag automata that use significantly less space than the original indexes. Even better, if we can identify the equivalence of states even before constructing the subautomata, then we can save significant amount of time in construction of the automata.

Identification of equivalent states is somewhat harder in backtracking indexes, since the equivalence of two states cannot be determined just by inspecting the subautomata rooted at these states. This is because the states inspected when backtracking out of the two subautomata may be different. As such, sharing of equivalent states has been studied primarily in the context of deterministic automata.

16.6.1. Issues and trade-offs

- Sharing can lead to significant reduction in space usage
- Direct construction of minimal automata (i.e., without first constructing unminimized automata) is crucial
- No significant negative effects

16.7. Factoring computation of substitutions

In the model described so far, the primitive steps in the retrieval operation consisted of checking whether the query term had a specific nonvariable symbol at a position. For collapsed tries, this operation generalizes to checking multiple positions. If this is to be extended for indexing nonlinear terms, one may add more powerful operations to the index, specifically, the operation of testing for unifiability of two variables. Such operations can be used to perform consistency checks among substitutions, especially in the context of problems such as simultaneous unification, multiliteral subsumption, etc. They also provide a convenient way to factorize the substitutions themselves. Such extensions have been studied in the context of several indexing techniques, although a majority of techniques known to date restrict themselves to linear terms. Even when a technique deals with nonlinear indexed terms, the more complex operations for consistency checking (of substitutions) are typically performed after all other operations.

16.7.1. Issues and trade-offs

- Treats indexing and computation of substitutions as a single process, rather than as separate phases, which avoids re-examination of symbols
- Shares not only common matching operations but also common variable substitutions, which leads to more efficient handling of nondeterminism.
- Can make an imperfect filtering technique into a perfect one.

16.8. Prioritized indexing

One way to deal with priorities is to ignore them completely during the indexing phase – instead, we identify all compatible terms using indexing, and then eliminate all except the terms with maximal priorities. In many applications, this may be unacceptable, since we may generate a large intermediate set of candidate terms first before eliminating most of them in the second step. Therefore, an indexing approach that only computes the terms with maximal priority is desirable. Among the indexing techniques described through the rest of the paper, some (e.g., adaptive pattern matching and unification factoring) methods incorporate priorities into the index, whereas other (e.g., path indexing) handles them at retrieval time.

16.8.1. Issues and trade-offs

- Construction vs. retrieval time — handling priorities makes the index construction more expensive, but retrieval is accelerated, e.g., in adaptive automata, the need to handle priorities increases construction time as well as size of the automata, but in an application where retrieval time is most important, this is a good trade-off.
- Priorities may make it possible to design (more) optimal algorithms, e.g., in unification factoring, optimal automata can be developed for linearly ordered indexed terms, but not for the unprioritized case; in adaptive automata, priority information can be used to build optimal automata for classes of terms for which no optimal automata exists in the unprioritized case.
- It is sometimes possible to perform the priority-related operations efficiently at runtime. For instance, in dynamic path indexing, path automata are con-

structed without taking priority into account, but at retrieval time, we compute the elements of this set in an order consistent with the priorities. When efficient methods exist for handling priority related operations at retrieval time, it may not be worth the additional cost of dealing with priorities in the index construction and maintenance operations.

16.9. Summary of indexing techniques

In this section, we summarize many known indexing methods in terms of their elements. We then discuss the benefits and disadvantages of each of these components. This discussion enables us to understand many known indexing methods as instances of a generic approach based on string-matching. It also makes it easier to compare and contrast known approaches.

Based on the terminology developed so far, we can classify the known term indexing techniques as shown in Table 1.

17. Indexing in practice

17.1. Logic programming and deductive databases.

Absence of indexing in logic programming systems will result in unnecessary backtracking, leading to poor performance. Almost all systems use some form of indexing. Specifically:

- Most logic programming systems use the outermost symbol of the first argument for indexing.
- The ALS system uses full indexing as described in [Chen et al. 1992].
- The XSB system [Sagonas, Swift, Warren, Freire and Rao 1997] uses unification factoring described in [Dawson, Ramakrishnan, Skiena and Swift 1996].
- The SICStus Prolog system [Int 2000] allows one to specify the positions in terms on which indexing is to be performed.

Experimental results show that unification factoring is a practical technique that can achieve substantial speedups for logic programs, while requiring no changes in the WAM. The speedups observed may be even more substantial when unification factoring is applied to programs which are themselves produced by transformations. For instance, the HiLog transformation [Cheng, Kifer and Warren 1993] increases the declarativeness of programs by allowing unification on predicate symbols. If implemented naively, however, HiLog can cause a decrease in efficiency for clause access. Experiments have shown that unification factoring can lead to speedups of 3 to 4 on HiLog code.

technique name	perfect filtering	prior-ities	number of strings	adapt-ive	determ-inistic	pruned/ collapsed	fail link/ dags	subst. factoring	retrieval condition
path indexing	NL	N	≥ 1	N	Y	N	N/A	N	all
automata-driven	NL	Y	≥ 1	N	Y	N	Y	N	*unif*
discrimin-ation tree	Y,NL	N	1	N	N	N	N	N	all
adaptive automata	NL	Y	1	Y	Y	N	Y	N	*gen*
abstraction trees	Y	N	1	Y	N	Y	N	Y	all
unification factoring	Y	Y	1	Y	N	Y	N	Y	*unif*
threaded tries	Y	Y	1	N	N	Y	Y	Y	*unif*
AC-discrim ination tree	NL	N	≥ 1	N	N	N	N	N	*gen*
AC-path indexing	N	N	≥ 1	N	Y	N	N/A	N	all
code tree	Y	N	1	Y,N	N	N	Y	Y	subsumption

Here NL means modulo nonlinearity, i.e., the retrieval is being done without any regard to variables that may be occur multiple times in the indexed terms or the query term. In other words, the filtering performed would be perfect if all the indexed terms and the query term were linear.

Table 1: Classification of term indexing techniques

17.2. Functional programming

Adaptive indexing is essential for doing lazy evaluation based on Huet-Levy style [Huet and Levy 1991]. Two systems that use such techniques are: Equals [Kaser, Pawagi, Ramakrishnan, Ramakrishnan and Sekar 1992, Kaser, Ramakrishnan, Ramakrishnan and Sekar 1997] and Gaml [Maranget 1992]. Some functional languages such as Haskell use a variation of standard and deterministic discrimination trees [Wadler 1987].

17.3. Theorem provers

In this section we overview the data structures and indexing techniques used by the modern resolution-based theorem provers. The information about particular provers is kindly provided[10] by Hans de Nivelle, Thomas Hillenbrand, Bill McCune, Robert Nieuwenhuis, Alexandre Riazanov, Stephan Schulz, and Christoph Weidenbach.

17.3.1. BLIKSEM

BLIKSEM uses imperfect *substitution trees* for forward subsumption and forward demodulation. For forward subsumption, only one term of each clause is stored in the tree.

For backward subsumption and demodulation BLIKSEM uses an interesting trick. Essentially, no special algorithms for backward subsumption or demodulation are implemented. But occasionally all clauses are rechecked, as if they were just generated, with the forward subsumption and demodulation checks.

For unification no indexing is used.

17.3.2. E

- *Perfect discrimination trees with weight constraints* for forward subsumption and forward demodulation. In addition, for forward demodulation age constraints are used.[11] For nonunit forward subsumption the use of discrimination trees gives an imperfect indexing technique, so for the nonunit case E uses sequential search with various prefilters (weight, class, individual literal matches).
- For backward demodulation E uses sequential search on *perfectly shared term structures* with reducibility markers and weight filtering.[12]

[10] In alphabetical order. The list of provers is also given in the alphabetical order.

[11] Experiments described in [Riazanov and Voronkov 2000*b*] have shown no gain of efficiency of the use of weight constraints for code trees as implemented in VAMPIRE.

[12] This is only used to find clauses which have to be eliminated from the set of processed clauses (because a term in a maximal or selected literal can be rewritten in a way that makes reprocessing necessary). Also note that during the search, normal form dates of irreducible subterms are updated, so that the age constraints of the PDTs make interreduction and actual rewriting very cheap. Interreduction as well as the actual rewriting are then performed using the perfect discrimination trees.

- For backward subsumption sequential search with prefilters as above is used. Unification is also implemented using sequential search, but the unification algorithm is equation-solving optimized for early mismatch detection.

17.3.3. Fiesta

Until recently, for all indexing algorithms Fiesta *substitution trees* were used. Currently, *context trees* are under implementation.

17.3.4. Otter

- *Discrimination trees* are used for forward subsumption and forward demodulation.
- *Path indexing* is used for backward subsumption, backward demodulation, and unification.

Multiliteral operations in Otter implement imperfect filtering, since the index contains literals, not clauses. However, the substitution computed during indexing on the literal level is used to complete the retrieval operation on the clause level.

17.3.5. Spass

Spass uses *substitution trees* for all indexing operations: forward subsumption and demodulation, backward subsumption and demodulation, and unification. For all operations perfect filtering is implemented and the retrieval operations deliver always the binding representing the substitution. The binding can then be used, without being applied to any term, to e.g., to verify ordering restrictions.

17.3.6. Vampire

- *Partially adaptive code trees* for forward subsumption and forward demodulation. In the case of forward demodulation, for nonoriented unit equalities the index also contains precompiled ordering constraints.
- *Path indexing* for backward subsumption and backward demodulation. The algorithm for retrieval uses optimization based on *database joins* and *skip lists*.
- *Perfectly shared terms* for storing terms occurring in kept clauses.
- *Flatterms* for representing terms occurring in temporary clauses.

17.3.7. Waldmeister

For unification, forward demodulation, and forward subsumption Waldmeister employs *perfect discrimination trees* with the pruning refinement that every branch leading to only one leaf node is shrunk.

At least in the unit equational case, keeping the clauses in SOS permanently normalized does not pay off.[13] So Waldmeister spends most of its runtime in the forward operations, especially in demodulation and subsumption; and backward subsumption and demodulation are implemented by sequential search.

[13]Thomas Hillenbrand, private communication.

18. Conclusion

Efficient indexing techniques have been developed for many retrieval operations arising in logic and functional programming, and theorem proving. In spite of this, important improvements of the existing indexing techniques are still possible and needed, and other techniques for previously not considered retrieval conditions need to be developed.

However, there are research directions in term indexing that need to be further investigated. In this section we sketch some possible research directions.

18.1. Comparison of indexing techniques

Until recently, it was difficult to compare the practical efficiency of term indexing methods in a unified framework. Previous comparisons suffered from two major problems. First, experimental results about performance of indexing technique were designed in such a way so that it was impossible to compare these results with a different implementation. As a result, superiority of a particular indexing technique was demonstrated by comparing a tightly coded implementation of this technique with nonoptimized experimental implementations of other techniques by the same person. Second, benchmarks for comparison were often nonrepresentative, for example, randomly generated.

A practical comparison method *COMPIT* (a method for *COMP*aring *I*ndexing *T*echniques) was described in a recent paper by Nieuwenhuis et al. [2001]. The idea of this method is the following. First, benchmarks for the method can be generated by running different provers. The method used for creating such benchmarks for a given prover is to add instructions making the prover write to a log file a trace each time an operation on the index takes place, and then run it on the given problem. For example, each time a term t is inserted (deleted, unified with), a trace like $+t$ (resp. $-t$, ut) is written to the file. Moreover, we require to store the traces along with information about the result of the operation (e.g., success/failure), which allows one to detect cases of incorrect behaviour of the indexing methods being tested.

The main part of the evaluation process is to test a given implementation of indexing on such a benchmark file. This given implementation is assumed to provide operations for querying and updating the indexing data structure, as well as a translation function for creating terms in its required format from the benchmark format. In order to avoid overheads and inexact time measurements due to translations and reading terms from the disk, the evaluation process first reads a large block of traces, storing them in main memory. After that, all terms read are translated into the required format. Then time measuring is switched on, and a loop is started which calls the corresponding sequence of operations, and time is turned off before reading the next block of traces from disk, and so on.

The method was applied in Nieuwenhuis et al. [2001] to compare the implementations of content trees in Dedam [Ganzinger et al. 2001], partially adaptive code trees in Vampire [Riazanov and Voronkov 2000*b*], and perfect discrimination trees in Waldmeister, for the retrieval of generalizations.

Based on this experiments the library *LIBERTI* (*LI*rary of *B*enchmarks for *E*fficient *R*etrieval and *T*erm *I*ndexing) was created. When benchmarks for other retrieval conditions are added to the library, we will have more empirical evidence about the comparative efficiency of various term indexing methods.

18.2. Indexing in presence of constraints

Constraints have two major uses in automated deduction. *Orderings constraints* are used for restricting the applicability of inference rules and eliminating redundant clauses, see [Bachmair and Ganzinger 2001, Nieuwenhuis and Rubio 2001] (Chapters 2 and 7 of this Handbook). Constraints are also used to present knowledge about built-in domains, for example integers or finite domains, see [Bockmayr and Weispfenning 2001] (Chapter 12).

When ordering constraints are used, it is often the case that after term retrieval an ordering constraint should be checked; and when the constraint is not satisfied, the retrieval process continues. Checking ordering constraints may be expensive, so it is desirable to built-in constraint checking in the retrieval process. The first step towards building-in constraint checking was done in Vampire: when retrieval of instances is used to implement backward demodulation, ordering constraints are compiled. When retrieval is finished, the compiled constraint is checked against the substitution that is computed as the result of retrieval.

18.3. Other retrieval conditions

There are several important retrieval conditions not covered in this paper but important enough for efficient implementation of theorem provers. Some examples are

- Retrieval conditions for multiliteral clauses, for example, forward and backward subsumption, and simultaneous unification. Some results and techniques appear in [Voronkov 1995] for forward subsumption and in [de Nivelle 1998] for simultaneous unification.
- Developing indexing methods for subterms. Bottom-up techniques for this problem is reported in [Hoffmann and O'Donnel 1982]. While some preliminary work based on top-down traversal appears in [Ramesh and Ramakrishnan 1992, Ramesh et al. 1994] there is still a need for faster methods.

Bibliography

Aït-Kaci H. [1991], *Warren's Abstract Machine: a Tutorial Reconstruction*, The MIT Press.

Bachmair L., Chen T. and Ramakrishnan I. [1993], Associative-commutative discrimination nets, *in* M.-C. Gaudel and J.-P. Jouannaud, eds, 'Proceedings of the 4th International Joint

Conference on Theory and Practice of Software Development (TAPSOFT)', Vol. 668 of *Lecture Notes in Computer Science*, Springer Verlag, Orsay, France, pp. 61–74.

Bachmair L., Chen T., Ramakrishnan I., Anantharaman S. and Chabin J. [1995], Experiments with associative-commutative discrimination nets, *in* C. Mellish, ed., 'International Joint Conference on Artificial Intelligence', Vol. 1, Montréal, pp. 348–355.

Bachmair L. and Ganzinger H. [2001], Resolution theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 2, pp. 19–99.

Benanav D., Kapur D. and Narendran P. [1987], 'Complexity of matching problems', *Journal of Symbolic Computation* **3**, 203–216.

Bockmayr A. and Weispfenning V. [2001], Solving numerical constraints, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 12.

Chen T., Ramakrishnan I. and Ramesh R. [1992], Multistage indexing algorithms for speeding Prolog execution, *in* K. Apt, ed., 'Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP-92)', MIT Press, pp. 639–653. Revised version [Chen et al. 1994].

Chen T., Ramakrishnan I. V. and Ramesh R. [1994], 'Multistage indexing algorithms for speeding Prolog execution', *Software — Practice and Experience* **24**(12), 1097–1119.

Chen W. and Warren D. S. [1996], 'Tabled evaluation with delaying for general logic programs', *Journal of the ACM* **43**(1), 20–74.

Cheng W., Kifer M. and Warren D. [1993], 'HILOG: a foundation for higher-order logic programming', *Journal of Logic Programming* **15**(3), 187–230.

Christian J. [1989], Fast Knuth-Bendix completion: A summary, *in* N. Dershowitz, ed., 'Proceedings of the 3rd International Conference on Rewriting Techniques and Applications', Vol. 355 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 551–555.

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

Comer D. and Sethi R. [1976], Complexity of trie index construction (extended abstract), *in* '17th Annual Symposium on Foundations of Computer Science', IEEE, Houston, Texas, pp. 197–207.

Dawson S., Ramakrishnan C. and Ramakrishnan I. [1995], Design and implementation of jump tables for fast indexing of logic programs, *in* M. V. Hermenegildo and S. D. Swierstra, eds, 'Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95', Vol. 982 of *Lecture Notes in Computer Science*, Springer Verlag, Utrecht, The Netherlands, pp. 133–150.

Dawson S., Ramakrishnan C., Ramakrishnan I., Sagonas K., Skiena S., Swift T. and Warren D. [1995], Unification factoring for efficient execution of logic programs, *in* 'ACM Symposium on Principles of Programming Languages', ACM Press, pp. 247–258.

Dawson S., Ramakrishnan C., Skiena S. and Swift T. [1996], 'Principles and practice of unification factoring', *ACM Transactions on Programming Languages and Systems* **18**(5), 528–563.

de Nivelle H. [1998], 'An algorithm for the retrieval of unifiers from discrimination trees', *Journal of Automated Reasoning* **20**(1/2), 5–25.

de Nivelle H. [2000], *Bliksem 1.10 User's Manual*, MPI für Informatik, Saarbrücken.

Dershowitz N. and Jouannaud J.-P. [1990], Rewrite systems, *in* J. Van Leeuwen, ed., 'Handbook of Theoretical Computer Science', Vol. B: Formal Methods and Semantics, North Holland, Amsterdam, chapter 6, pp. 243–309.

Eker S. M. [1995], 'Associative-commutative matching via bipartite graph matching', *Computer Journal* **38**(5), 381–399.

Ganzinger H., Nieuwenhuis R. and Nivela P. [2001], Context trees. Submitted to IJCAR.

Gottlob G. and Leitsch A. [1987], 'On the efficiency of subsumption algorithms', *Journal of the ACM* **32**(2), 280–295.

Graf A. [1991], Left-to-right tree pattern matching, *in* R. Book, ed., 'RTA'91', Vol. 488 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 323–334.

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

Graf P. [1995], Substitution tree indexing, *in* J. Hsiang, ed., 'Rewriting Techniques and Applications', Vol. 914 of *Lecture Notes in Computer Science*, pp. 117–131.

Graf P. [1996], *Term Indexing*, Vol. 1053 of *Lecture Notes in Computer Science*, Springer Verlag.

Greenbaum S. [1986], Input transformations and resolution implementation techniques for theorem-proving in first-order logic, PhD thesis, University of Illinois at Urbana-Champaign.

Henschen L. and Naqvi S. [1981], An improved filter for literal indexing in resolution systems, *in* '6th Int'l Joint Conference on Artifical Intelligence (IJCAI)', pp. 528–529.

Hewitt C. [1971], Description and theoretical analysis of Planner: A language for proving theorems and manipulating models in a robot, PhD thesis, Department of Mathematics, MIT.

Hillenbrand T., Buch A., Vogt R. and Löchner B. [1997], 'Waldmeister: High-performance equational deduction', *Journal of Automated Reasoning* **18**(2), 265–270.

Hoffmann C. M. and O'Donnel M. J. [1982], 'Pattern matching in trees', *Journal of the ACM* **29**(1), 68–95.

Huet G. and Levy J.-J. [1978], Computation in nonambiguous linear term rewriting systems, Technical Report 359, IRIA, Le Chesney, France. Revised version in [Huet and Levy 1991].

Huet G. and Levy J.-J. [1991], Computation in orthogonal rewriting systems, I and II, *in* J.-L. Lassez and G. Plotkin, eds, 'Computational Logic. Essays in Honor of Alan Robinson.', MIT Press, pp. 395–443. Earlier version appeared as [Huet and Levy 1978].

Int [2000], *Sictus Prolog User's Manual. Release 3.8.4*.

Kaser O., Pawagi S., Ramakrishnan C., Ramakrishnan I. and Sekar R. [1992], Fast parallel implementation of lazy languages — the EQUALS experience, *in* '1992 ACM Conference on Lisp and Functional Programming', ACM Press, pp. 335–344.

Kaser O., Ramakrishnan C., Ramakrishnan I. and Sekar R. [1997], 'EQUALS — a parallel implementation of a lazy language', *Journal of Functional Programming* **7**(2), 183–217.

Kennaway J. [1990], The specificity rule for lazy pattern matching in ambiguous term rewriting systems, *in* N. Jones, ed., 'ESOP'90, 3rd European Symposium on Programming', Vol. 432 of *Lecture Notes in Computer Science*, Springer Verlag, Copenhagen, Denmark, pp. 256–270.

Knuth D., Morris J. and Pratt V. [1977], 'Fast pattern matching in strings', *SIAM Journal of Computing* **6**(2), 323–350.

Kounalis E. and Lugiez D. [1991], Compilation of pattern matching with associative-commutative functions, *in* S. Abramsky and T. Maibaum, eds, 'Proceedings of the International Joint Conference on Theory and Practice of Software Development, volume 1: Colloquium on Trees in Algebra and Programming', number 493 *in* 'Lecture Notes in Computer Science', Springer Verlag, Brighton, U.K., pp. 57–73.

Laville A. [1987], Lazy pattern matching in the ML language, *in* K. Nori, ed., 'Foundations of Software Technology and Theoretical Computer Science, 7th Conference', Vol. 287 of *Lecture Notes in Computer Science*, Springer Verlag, Pune, India, pp. 400–419.

Laville A. [1988], Implementation of lazy pattern matching algorithms, *in* 'Proceedings of the European Symposium on Programming', Vol. 300 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 298–316.

Letz R., Schumann J., Bayerl S. and Bibel W. [1992], 'SETHEO: A high-performance theorem prover', *Journal of Automated Reasoning* **8**(2), 183–212.

Lugiez D. and Moysset J. [1993], Complement problems and tree automata in AC-like theories, *in* P. Enjalbert, A. Finkel and K. W. Wagner, eds, 'Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS'93)', Vol. 665 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 515–524.

LUSK E. [1992], Controlling redundancy in large search spaces: Argonne-style theorem proving through the years, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning. International Conference LPAR'92.', Vol. 624 of *Lecture Notes in AI*, St.Petersburg, Russia, pp. 96–106.

MARANGET L. [1992], Compiling lazy pattern matching, *in* 'ACM Conference on Lisp and Functional Programming', pp. 21–31.

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

MCCUNE W. [1994*a*], OTTER 3.0 reference manual and guide, Technical Report ANL-94/6, Argonne National Laboratory/IL, USA.

MCCUNE W. [1994*b*], OTTER 3.0 reference manual and guide, Technical Report ANL-94/6, Argonne National Laboratory.

MCCUNE W. AND WOS L. [1997], 'Otter—the CADE-13 competition incarnations', *Journal of Automated Reasoning* **18**(2), 211–220.

MOSER M., IBENS O., LETZ R., STEINBACH J., GOLLER C., SCHUMANN J. AND MAYR K. [1997], 'SETHEO and E-SETHEO—the CADE-13 systems', *Journal of Automated Reasoning* **18**, 237–246.

NELSON G. AND OPPEN D. [1980], 'Fast decision procedures based on congruence closure', *Journal of the ACM* **27**(2), 356–364.

NICOLAITA D. [1992], An indexing scheme for AC-equational theories, Technical report, Research Institute for Infomatics, Bucharest, Romania.

NIEUWENHUIS R., HILLENBRAND T., RIAZANOV A. AND VORONKOV A. [2001], Let's COMPIT: a method for COMParing Indexing Techniques in theorem provers, Preprint CSPP-11, Department of Computer Science, University of Manchester.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

NIEUWENHUIS R., RIVERO J. AND VALLEJO M. [1997], 'The Barcelona prover', *Journal of Automated Reasoning* **18**(2), 171–176.

NIEUWENHUIS R. AND RUBIO A. [2001], Paramodulation-based theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 7.

OHLBACH H. [1990], Abstraction tree indexing for terms, *in* 'Proceedings of the 9th European Conference on Artificial Intelligence', Pitman Publishing, London, pp. 479–484.

PUEL L. [1990], Compiling pattern matching by term decomposition, *in* 'ACM Conference on Lisp and Functional Programming', pp. 273–281.

RAMAKRISHNAN R. [1991], 'Magic templates: A spellbinding approach to logic programs', *Journal of Logic Programming* **11**(3 & 4), 189–216.

RAMAMOHANARAO K. AND SHEPHERD J. [1986], A superimposed codewords indexing scheme for very large Prolog databases, *in* E. Shapiro, ed., 'Proceedings of the Third International Conference on Logic Programming', Vol. 225 of *Lecture Notes in Computer Science*, Springer Verlag, London, pp. 569–576.

RAMESH R., RAMAKRISHNAN I. AND SEKAR R. [1994], Automata-driven efficient subterm unification, *in* P. S. Thiagarajan, ed., 'Foundations of Software Technology and Theoretical Computer Science, 14th Conference', Vol. 880 of *Lecture Notes in Computer Science*, Springer Verlag, Madras, India, pp. 288–299.

RAMESH R. AND RAMAKRISHNAN I. V. [1992], 'Nonlinear pattern matching in trees', *JACM* **39**(2), 295–316.

RAMESH R., RAMAKRISHNAN I. AND WARREN D. [1990], Automata-driven indexing of Prolog clauses, *in* 'Seventh Annual ACM Symposium on Principles of Programming Languages', San Francisco, pp. 281–290. Revised version [Ramesh, Ramakrishnan and Warren 1995].

RAMESH R., RAMAKRISHNAN I. AND WARREN D. [1995], 'Automata-driven indexing of Prolog clauses', *Journal of Logic Programming* **23**(2), 151–202.

Rao P., Ramakrishnan C. and Ramakrishnan I. [1996], A thread in time saves tabling time, *in* M. Maher, ed., 'Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming', MIT Press, pp. 112–126.

Riazanov A. and Voronkov A. [1999], Vampire, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in AI*, Trento, Italy, pp. 292–296.

Riazanov A. and Voronkov A. [2000*a*], Limited resource strategy in resolution theorem proving, Preprint CSPP-7, Department of Computer Science, University of Manchester.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

Riazanov A. and Voronkov A. [2000*b*], Partially adaptive code trees, *in* M. Ojeda-Aciego, I. de Guzmán, G. Brewka and L. Pereira, eds, 'Logics in Artificial Intelligence. European Workshop, JELIA 2000', Vol. 1919 of *Lecture Notes in AI*, Springer Verlag, Málaga, Spain, pp. 209–223.

Riazanov A. and Voronkov A. [2001*a*], An efficient algorithm for backward subsumption using path indexing and database joins, Preprint, Department of Computer Science, University of Manchester. To appear.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

Riazanov A. and Voronkov A. [2001*b*], Splitting without backtracking, Preprint CSPP-10, Department of Computer Science, University of Manchester.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

Rivero J. [2000], Data Structures and Algorithms for Automated Deduction with Equality, Phd thesis, Universitat Politècnica de Catalunya, Barcelona.

Sagonas K., Swift T., Warren D., Freire J. and Rao P. [1997], The XSB programmer's manual: version 1.7, Technical report, SUNY at Stony Brook.

Schulz S. [1999], System abstract: E 0.3, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Lecture Notes in AI, Trento, Italy, pp. 297–301.

Sekar R. C., Ramesh R. and Ramakrishnan I. V. [1995], 'Adaptive pattern matching', *SIAM Journal on Computing* **24**(6), 1207–1234.

Sekar R., Ramesh R. and Ramakrishnan I. [1992], Adaptive pattern matching, *in* W. Kuich, ed., 'Proceedings of the 19th International Colloquium on Automata, Languages and Programming', number 623 *in* 'Lecture Notes in Computer Science', Springer Verlag, Vienna, pp. 247–260. Revised version [Sekar et al. 1995].

Socher R. [1989], A subsumption algorithm based on characteristic matrices, *in* N. Dershowitz, ed., 'Rewriting Techniques and Applications, 3rd International Conference, RTA-89', Vol. 355 of *Lecture Notes in Computer Science*, Springer Verlag, Chapel Hill, North Carolina, USA, pp. 573–581.

Stickel M. [1988], 'A PROLOG technology theorem prover: Implementation by an extended Prolog compiler', *Journal of Automated Reasoning* **4**, 353–380.

Stickel M. E. [1989], The path-indexing method for indexing terms, Technical Report 473, SRI International, Menlo Park, California, USA.

Stickel M., Waldinger R. and Chaudhry V. [2000], *A Guide to SNARK*, SRI International.
URL: *www.ai.sri.com/hpkb/snark/tutorial*

Stickel M., Waldinger R., Lowry R., Pressburger T. and Underwood I. [1994], Deductive composition of astronomical software from subroutine libraries, *in* A. Bundy, ed., 'Automated Deduction — CADE-12. 12th International Conference on Automated Deduction', Vol. 814 of *Lecture Notes in AI*, Nancy, France, pp. 341–355.

Sutcliffe G. and Suttner C. [1998], 'The TPTP problem library — CNF release v. 1.2.1', *Journal of Automated Reasoning* **21**(2).

Tammet T. [1997], 'Gandalf', *Journal of Automated Reasoning* **18**(2), 199–204.

Verma R. M. and Ramakrishnan I. [1992], 'Tight complexity bounds for term matching problems', *Information and Computation* **101**(1), 33–69.

Voronkov A. [1994], An implementation technique for a class of bottom-up procedures, *in* M. Hermenegildo and J. Penjam, eds, 'Programming Languages Implementation and Logic Programming. 6th International Symposium, PLILP'94', Vol. 844 of *Lecture Notes in Computer Science*, Madrid, pp. 147–164.

Voronkov A. [1995], 'The anatomy of Vampire: Implementing bottom-up procedures with code trees', *Journal of Automated Reasoning* **15**(2), 237–265.

Wadler P. [1987], Efficient compilation of pattern matching, *in* S. P. Jones, ed., 'The Implementation of Functional Programming Languages', Prentice Hall, chapter 5.

Weidenbach C. [2001], Combining superposition, sorts and splitting, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 27.

Weidenbach C., Gaede B. and Rock G. [1996], Spass & flotter version 0.42, *in* M. McRobbie and J. Slaney, eds, 'Automated Deduction — CADE-13', Vol. 1104 of *Lecture Notes in Computer Science*, New Brunswick, NJ, USA, pp. 141–145.

Wise M. and Powers D. [1984], Indexing Prolog clauses via superimposed codewords and field encoded words, *in* 'Proceedings of the International Symposium on Logic Programming', Computer Society Press, pp. 203–210.

Wos L. [1992], 'Note on McCune's article on discrimination trees', *Journal of Automated Reasoning* **9**(2), 145–146.

Index