Fast Term Indexing with Coded Context Trees [*]

Harald Ganzinger (hg@mpi-sb.mpg.de)
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany.

Robert Nieuwenhuis (roberto@lsi.upc.es) and Pilar Nivela (nivela@lsi.upc.es)
Technical University of Catalonia, Dept. LSI,
Jordi Girona 1, 08034 Barcelona, Spain

Abstract. Indexing data structures have a crucial impact on the performance of automated theorem provers. Examples are discrimination trees, which are like tries where terms are seen as strings and common prefixes are shared, and substitution trees, where terms keep their tree structure and all common contexts can be shared. Here we describe a new indexing data structure, called context trees, where, by means of a limited kind of context variables, also common subterms can be shared, even if they occur below different function symbols. Apart from introducing the concept, we also provide evidence for its practical value. We show how context trees can be implemented by means of abstract machine instructions. Experiments with matching benchmarks show that our implementation is competitive with tightly coded current state-of-the-art implementations of the other main techniques. In particular space consumption of context trees is significantly less than for other index structures.
Keywords: term rewriting, automated deduction.

1. Introduction

Indexing data structures have a crucial impact on the performance of theorem provers. The indexes have to store a large number of terms and to support the fast retrieval, for any given *query* term t, of all terms in the index satisfying a certain relation with t, such as matching, unifiability, or syntactic equality. Indexing for matching, where, to check for forward redundancy, one searches in the index for a generalisation of the query term, is well-known to be the most limiting bottleneck in practice. Another aspect which is becoming more and more crucial is memory consumption. During the last years processor speed has been growing faster than memory capacity and one may assume that this gap will become wider in the coming years. At the same time memory access bandwidth is also becoming an important bottleneck. Excessive memory consumption leads to more cache faults, which become the

[*] The second and third author are partially supported by the Spanish CICYT project HEMOSS ref. TIC98-0949-C02-01.

dominant factor for time, instead of processor speed. Therefore, in what follows we will mainly focus on matching retrieval operations and, apart from retrieval speed, also on memory consumption.

One important aspect makes indexing techniques in theorem proving essentially different from indexing in other contexts such as functional or logic programming: the index is subject to insertions and deletions. If the index were stable and only of moderate size one could afford to spend substantial compilation time in order to represent it, for instance, as a deterministic tree automaton of possibly exponential size in the size of the patterns. Such an automata representation would guarantee fast retrieval times linear in the size of the query. In theorem proving applications the index can become very large and is subject to dynamic changes. Therefore, during the last two decades a significant number of results on new specific indexing techniques for theorem proving have been published and applied in different provers. The currently best-known and most frequently used indexing techniques for matching are *discrimination trees* (Christian, 1993; McCune, 1992), the compiled variant of discrimination trees, called *code trees* (Voronkov, 1995), and *subsitutution trees* (Graf, 1995).

Discrimination trees are like tries where terms are viewed as strings and where common prefixes are shared. A substitution tree has, in each node, a substitution, a list of pairs $x_i \mapsto t$ where each x_i is an *internal* variable and t is a term that may contain other internal variables as well as *external* variables which are the variables in the terms to be stored.

EXAMPLE 1. *The two terms $f(a, g(x), h(y))$ and $f(h(b), g(y), h(y))$ can be stored in a substitution tree and discrimination tree, respectively, as shown:*

substitution tree:

$$\begin{tikzcd}[column sep=tiny] & x_1 = f(x_2, g(x_3), h(y)) \arrow[dl] \arrow[dr] & \\ x_2 = a,\ x_3 = x & & x_2 = h(b),\ x_3 = y \end{tikzcd}$$

discr. tree:

$$\begin{tikzcd}[column sep=tiny] & f \arrow[dl] \arrow[dr] & \\ a \arrow[d] & & h \arrow[d] \\ g \arrow[d] & & b \arrow[d] \\ \vdots & & \vdots \end{tikzcd}$$

In a substitution tree all terms $x_1\sigma$ are stored such that σ is the composition of the substitutions on some path from the root to a leaf of the tree. In the example, after inserting the first term in an empty substitution tree we obtain the single node $x_1 = f(a, g(x), h(y))$. When inserting the second term, internal variables are placed at the positions of dis-

agreement, and children are created with the "remaining" substitutions of both. Therefore all common contexts can be shared. □

EXAMPLE 2. *It is clear that the additional sharing in substitution trees avoids repeated work, which is in fact the key idea behind all indexing techniques. Assume one has two terms $f(c, x, t)$ and $f(x, c, t)$ in the index, and a query $f(c, c, s)$, where s and t are terms such that s is not an instance of t. Then two attempts to match s against t will be made in a discrimination tree, and only one in a substitution tree.* □

Here we describe a new indexing data structure, called *context trees*, where, by means of a limited kind of context variables, certain common subterms can be shared, even if they occur below different function symbols. Roughly, the idea is that $f(s)$ and $g(s, t)$ can be represented as $F(s, t)$, with children $F = f$ and $F = g$. Function variables such as F stand for single function symbols only (although extensions to allow for more complex forms of second-order terms are possible).

EXAMPLE 3. *Assume one has three terms $h(x, f(t))$, $h(x, g(t))$, and $h(b, f(t))$ in the index. Then, in a discrimination tree, t will occur three times. In a substitution tree, we will have:*

$$\begin{array}{c} x_1 = h(x_2, x_3) \\ \begin{array}{cc} x_2 = x & x_2 = b,\ x_3 = f(t) \\ \begin{array}{cc} x_3 = f(t) & x_3 = g(t) \end{array} & \end{array} \end{array}$$

and with a query $h(b, f(s))$, the terms s and t will be matched twice against each other (at the leftmost and rightmost leaves). In a context tree, the term t occurs only once:

$$\begin{array}{c} x_1 = h(x_2, F(t)) \\ \begin{array}{cc} x_2 = x & x_2 = b,\ F = f \\ \begin{array}{cc} F = f & F = g \end{array} & \end{array} \end{array}$$

and if s does not match t, the failure with the query $h(b, f(s))$ will be found at the root. □

In addition to proposing the concept of context trees, in this paper we will also provide evidence for its practical value. We show how they can be adequately implemented, and give empirical results on well-designed experiments.

This paper is structured as follows. Section 2 introduces some basic concepts of indexing, discrimination trees and substitution trees.

In Section 3 we first outline some problems with direct implementations of context trees. In order to solve theses problems, we then consider terms built from a single pairing constructor and constants. In Section 4 we describe a first conceptual implementation based on these Curry terms and an extension of substitution trees in two ways: (i) we add the concept of equality constraints and (ii) we merge *internal* and *external* variables into a single kind of variables. After that, in Section 5 we show how this conceptual version can be implemented in a very compact and fast way by means of abstract machine instructions in the style of (Riazanov and Voronkov, 2000). In these instructions, registers correspond to variables of the tree, and, as we will see, assigning registers according to the positions of each variable in the terms is the crucial idea for efficiency.

Experiments with matching are escribed in Section 6. They show that our implementation is competitive with tightly coded state-of-the-art implementations, namely the implementation of discrimination trees of the Waldmeister prover (Hillenbrand et al., 1997) and the code trees of the Vampire prover (Voronkov, 1995).

For the experiments, we adopted the methods for evaluation of indexing techniques described in (Nieuwenhuis et al., 2001): (i) we use 30 very large benchmarks containing the exact sequence of (update and retrieval) operations on the matching index that take place when running three well-known state-of-the-art provers on a diverse set of problems; (ii) comparisons are made with the discrimination tree implementation of the Waldmeister prover (Hillenbrand et al., 1997), and the code trees of the Vampire prover (Voronkov, 1995), as provided by their own implementors using the test driver of (Nieuwenhuis et al., 2001).

Finally, Section 7 concludes and describes some promising directions for future work.

2. Discrimination trees and substitution trees

Discrimination trees can be made very efficient if query terms are linear (as the terms in the trees are). Usually, queries are the so-called *flatterms* of (Christian, 1993), which are linked lists with additional pointers to jump over subterms t when a variable of the index gets instantiated by t.

In *standard discrimination trees*, all variables are represented by a single variable symbol $*$, so that different terms such as $f(x, y)$ and $f(x, x)$ are both represented by $f(*, *)$, and the corresponding path in the tree is common to both. This increases the amount of sharing, and also the retrieval speed, because the low-level operations (basically symbol comparison and variable instantiation) are very simple. But it is only a *prefilter*: once a possible match has been found, additional equality tests have to be performed between the query subterms by which the variables of terms like $f(x, x)$ have been instantiated. Nodes are usually arrays of pointers indexed by the function symbols, plus one additional pointer for $*$. If, during matching, the query symbol currently treated is f, then one can directly jump to the child for f, if it exists, or to the one of $*$. Especially for larger signatures, this representation of nodes leads to high memory consumption. Note that the case where children for both f and $*$ exist is the only situation where backtracking points are created.

In *perfect discrimination trees*, variables are not collapsed into a single symbol. Instead, nodes of different sizes exist: apart from the function symbols, each node can have a child for any of the variables that already occurred along the path in the tree, plus an additional child for a possible new variable. Hence even more memory is needed in this approach. Also there is less sharing in the index. On the other hand, the equality tests are not delayed (which is good according to the first-fail prinicple; see also below), all matches found are correct and no later equality tests are needed. The Waldmeister prover (Hillenbrand et al., 1997) uses these perfect discrimination trees for matching.

2.1. Implementation techniques for substitution trees

Let us now consider substitution trees in more detail. They were introduced by Peter Graf (Graf, 1995), who also developed an implementation that is still used in the Spass prover (Weidenbach, 1997). A more efficient implementation was given in the context of the Dedam (Deduction abstract machine) kernel of data structures (Nieuwenhuis et al., 1997).

As for discrimination trees, it is important to deal with an adequate representation of query terms. In Dedam, Prolog-like terms are used: each term $f(t_1, \ldots, t_n)$ is represented by $n + 1$ contiguous *heap cells* with a *tag* and an *address* field:

a	f	
$a+1$	`ref`	a_1
	$\vdots$	$\vdots$
$a+n$	`ref`	a_n

where each address field a_i points to the subterm t_i, and (uninstantiated) variables are `ref`'s pointing to themselves. In this setting, contiguous heap cell blocks of different sizes co-exist, and traversal of terms requires controlling arities. Term-to-term operations like matching or unification only instantiate self-referencing `ref` positions. If these instantiated positions are pushed on a stack, called the *refstack*, then undoing the operation amounts to restoring the positions in the refstack to self-references again.

Substitutions in substitution trees can be represented as pairs of heap addresses; each right hand side points to a term; each left hand side points to an internal variable (i.e., a self-ref position) occurring exactly once in some term at the right hand side of a substitution along the path to the root.

The basic idea for all retrieval operations (finding a term, matching, unification) in substitution trees is the same: one instantiates the internal variable x_1 at the root with the query term, and traverses the tree, where at each visited node with a substitution $y_1 = t_1, \ldots, y_n = t_n$, one performs the basic term-to-term operation (syntactic equality, matching, unification) between each (already instantiated) y_i and its corresponding t_i. The term-to-term operations only differ in which variables are allowed to be instantiated, and which variables are considered as constants: for finding terms (syntactic equality), only the internal `ref`'s (called `intref`) can be instantiated; for matching, also the external `ref`'s of the index (but not of the query); for unification, all `ref`'s can be instantiated.

Upon failure, backtracking occurs. After successfully visiting a node, before continuing with its first child, its next sibling is stored in the *backtracking stack*, together with the current height of the refstack. Therefore, for backtracking, one pops the next node to visit from the backtracking stack, together with its corresponding refstack height, and restores all `ref` positions above this height. A failure occurs when trying to backtrack on an empty backtracking stack.

2.2. Substitution trees for matching

In Dedam (Nieuwenhuis et al., 1997) a special version of substitution trees for matching has been developed, which is about three times faster than the general-purpose implementation in Spass and Dedam.

EXAMPLE 4. *Suppose the query is of the form $f(s,t)$ and consider a substitution tree with the two terms $f(x,x)$ and $f(a,x)$: the root is $x_1 = f(x_2, x)$, with children $x_2 = x$ and $x_2 = a$. When matching, at the root x_2 gets instantiated with s, and x with t; then, at the leftmost child, the terms s and t are matched against each other. Note that one has to keep track of whether or not x has already been instantiated, i.e., one has to keep a refstack.* $\square$

The idea for improving this procedure is similar to the one of the *standard* variant of discrimination trees: external variables are all considered to be different. For instance, the term $f(x,y,x,y,y)$ is inserted as $f(x_2,x_3,x_4,x_5,x_6)$ with the associated information that x_2 and x_4 are in the same equivalence class, and that x_3, x_5, and x_6 are in the same class. But in substitution trees the advantages are more effective: the refstack becomes unnecessary (and hence also the information about its height in the backtracking stack), because one can always override the values of the internal and external variables and restauration becomes unnecessary. Matching operations between query subterms, like s and t in the previous example, are replaced by a cheaper syntactic equality test of *equality constraints* like $x_2 = x_4, \quad x_3 = x_5, \quad x_3 = x_6$. In the Dedam implementation, these equality constraints are checked at the leaves of the substitution tree.

3. Context trees

We start by illustrating the increased amount of sharing in context trees as intuitively described in Section 1 compared with substitution trees.

EXAMPLE 5. *Assume in a context tree we have a subtree T below a node $x_i = f(x_j,t)$ (depicted below at the left) where we have to insert $x_i = g(s,t,u)$. Then the common part is $x_i = F(x_j,t,u)$, the specific parts are $\{F = f\}$ and $\{F = g,\ x_j = s\}$, respectively, and we obtain:*

$$
\begin{array}{c}
/ \\
x_i = f(x_j,t) \\
| \\
T
\end{array}
\qquad\qquad
\begin{array}{ccc}
 & / & \\
 & x_i = F(x_j,t,u) & \\
/ & & \backslash \\
F = f & & F = g,\ x_j = s \\
| & & \\
T & &
\end{array}
$$

During retrieval on the new tree, the term-to-term operations have to be guided by the arities of the query: if x_i is instantiated with a query term *headed with f, when arriving at the node $x_i = F(x_j, t, u)$, then, since the arity of f is 2, one can simply ignore the term u of this common part.* □

It is not difficult to see that, with the restricted kind of function variables F that stand for single function symbols, the common part of two terms s and t, as in substitution trees, will contain the entire common context, and additionally also those subterms u that occur at the same position p in both terms, that is, for which $u = s|_p = t|_p$.

EXAMPLE 6. *The common part of the two terms $f(g(b,b),a,c)$ and $h(h(b,c),d)$ is $F(G(b,x_1),x_2,x_3)$. Indeed, the subterm b at position* 1.1 *is the only term occurring at the same position in both terms.* □

To implement context trees for matching by an extension of the specialised substitution trees for matching requires to deal with the specific properties of the context variables.

EXAMPLE 7. *Consider again Example 5. The term $f(x_j, t)$ consists of three contiguous heap cells. The first contains f, the second is an* `intref` *corresponding to x_j, and the third is a* `ref` *pointing to the subterm t. Initially, in the subtree T below that node, along each path to a leaf x_j appears once as a left hand side in a substitution. After inserting $x_i = g(s,t,u)$, the common part is $x_i = F(x_j,t,u)$, and the new term $F(x_j,t,u)$ needs four contiguous heap cells instead of three.*

A serious implementation problem now is that, if we allocate a new block of size four, all left hand sides pointing to x_j in the subtree T have to be changed to point to the new address of x_j. □

3.1. Context trees through Curry terms

A simple solution to the problem described in the previous example would be to always use blocks corresponding to the maximal arity of symbols, but this is too expensive in memory consumption. Here we propose a different solution, which is conceptually appealing and at the same time turns out to be very efficient since it completely avoids the need for checking arities. We suggest to represent all terms in Curry form. Curry terms are formed with a single binary *apply* symbol, written here "·", and all other function symbols are considered (second-order) constants to be treated much like their first-order counterparts. This idea is standard in the context of functional programming, but, surprisingly, does not seem to have been considered for term indexing data structures for automated deduction before.

EXAMPLE 8. *Consider again the terms of Example 5, where we saw that one can share the term t in $f(x_j, t)$ and $g(s, t, u)$ through $F(x_j, t, u)$. In Curry form, these terms become $\cdot(\cdot(f, x_j), t)$ and $\cdot(\cdot(\cdot(g, s), t), u)$ and the t cannot be shared. But in the Curry form the same amount of sharing exists: still all arguments that are in the same position are shared, assuming that* positions are counted from right to left. *Consider the arguments of the same terms in reverse order. The we have $f(t, x_j)$ and $g(u, t, s)$, which in Curry form become $\cdot(\cdot(f, t), x_j)$ and $\cdot(\cdot(\cdot(g, u), t), s)$. The common part, which was $F(u, t, x_j)$, can be computed on the Curry terms exactly as it was done for common contexts of first-order terms in substitution trees. In the example we get $\cdot(\cdot(x_k, t), x_j)$, where the remaining parts are $\{x_k = f\}$ and $\{x_k = \cdot(g, u),\ x_j = s\}$.*

It is not difficult to see that in this way one obtains exactly the same amount of sharing as with context variables: all common contexts and all subterms u that occur at the same position in both terms (but remember: if positions are computed from right to left; for instance, the shared t in $f(t, x_j)$ and $g(u, t, s)$ is at position 2 in both terms). □

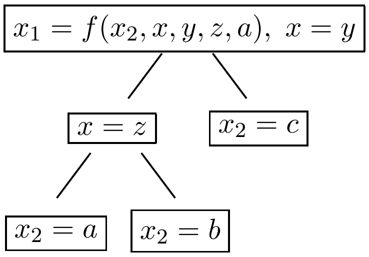
An important additional advantage is that the basic algorithms do not depend on the arities of the symbols anymore. Moreover, it is obviously not necessary to store any apply symbols.

4. Conceptual Implementation

In this section we give a describe an implementation of context trees at the conceptual level, without considering abstract machine instructions yet. C-like pseudo-code is given for the three basic operations: term-to-term matching, substitution-to-substitution matching, and the full context tree matching algorithm.

4.1. Equality constraints

In order to exploit the idea of equality constraints in its full power, it is important to perform the equality tests not only at the leaves, but as high up as possible in the tree without decreasing the amount of sharing. For example, if we have $f(a, x, x, x, a)$, $f(b, x, x, x, a)$, and $f(c, y, y, x, a)$, then the tree can be:

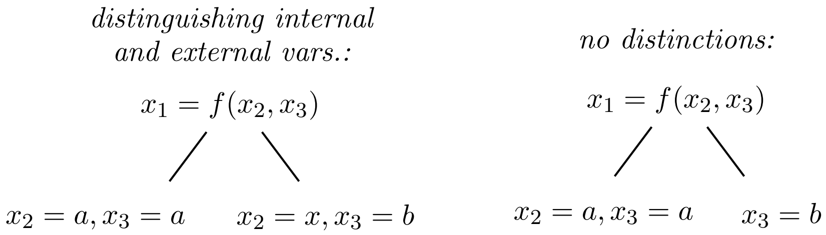


Note that placing the equality tests $x = y$ and $x = z$ in the leaves would frequently lead to repeated work during retrieval time. Also, according to the first-fail principle (which is strongly recommended in indexing techniques), it is important to impose strong restrictions like the equality of two whole subterms as soon as possible. Below we outline some details about our implementation of equality constraints, their evaluation during retrieval time and their creation during insertions by means of union-find data structures.

4.2. Internal vs external variables

After introducing equality constraints and Curry terms, in turns out that one can drop the distinction between internal and external variables that is usual in substitution trees. The idea is that any variable that is not instantiated when a leaf is reached represents an external variable. This leads to even more sharing in the index and increases matching retrieval speed, however, at the price of significantly more complex update operations (see below).

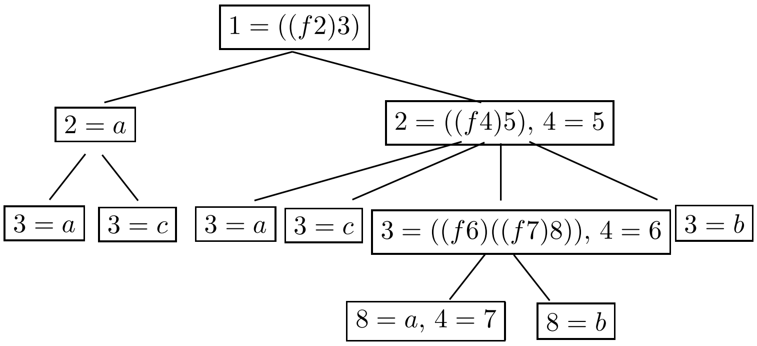
EXAMPLE 9. *If the tree contains the two terms $f(a, a)$ and $f(x, b)$, we have:*



Note that in the second tree the variable x_2 plays the role of an internal variable in the leftmost branch and of an external variable in the other one. □

In this setting, the term-to-term matching operation becomes:

```
int TermMatch(_Term query, _Term set){
  if (IsVar(set)) { Instantiate(set,query); return(1); }
```



1)	$f(a,a)$	$((fa)a)$
2)	$f(a,c)$	$((fa)c)$
3)	$f(f(0,0),a)$	$((f((f0)0))a)$
4)	$f(f(0,0),c)$	$((f((f0)0))c)$
5)	$f(f(0,0),f(0,f(0,a)))$	$((f((f0)0))((f0)((f0)a)))$
6)	$f(f(0,0),f(0,f(0,b)))$	$((f((f0)0))((f0)((f1)b)))$
7)	$f(0,b)$	$((f0)b)$

Figure 1. Context tree for the terms 1–7

```
  if (TopSymbol(query)!=TopSymbol(set)) return(0);
  if (IsApply(set))
    if ( !TermMatch( LeftChild(query),LeftChild(set) ) ||
         !TermMatch(RightChild(query),RightChild(set)) )
    return(0);
  return(1);}
```

4.3. Retrieval for Matching

In Fig. 1 we show a tree as it would have been generated in our implementation after inserting the seven terms given both in standard representation and Curry form, respectively. Variables are written as numbers. Note that the equality constraints $4 = 5$ and $4 = 6$ are shared among several branches. Using the term-to-term operations for equality and matching, the remaining code needed for matching retrieval on a context tree is very simple. One needs a function for checking the substitution of a context tree node during matching:

```
int SubstMatch(_Subst subst){
  while (subst){
    if (subst->IsEqualityConstraint)
      {if (!TermEqual(subst->lhs,subst->rhs)) return(0);}
    else
      {if (!TermMatch(subst->lhs,subst->rhs)) return(0);}
```

```
    subst = subst->next;}
  return(1);}
```

Finally, the general traversal algorithm of the tree is the one presented below (assuming that the root variable x_1 has already been instantiated with the query):

```
int CTreeMatch(_CTree tree){
   if (!SubstMatch(tree->subst))
      if (tree->nextSibling)
         return( CTreeMatch(tree->nextSibling) );
      else return(0);
   if ( tree->isLeaf ) return(1);
   if (!tree->nextSibling)
      return( CTreeMatch(tree->firstChild) );
   return( CTreeMatch(tree->nextSibling) ||
           CTreeMatch(tree->firstChild )  ); }
```

4.4. Updates

Updates are significantly more complex in context trees than in the standard substitution trees.

For insertion, one starts with a linearized term, together with a union-find data structure for keeping the information about the equivalence classes of the variables. For instance, the term $f(x, y, x, y, y)$ is inserted as $f(x_2, x_3, x_4, x_5, x_6)$ with the associated information that x_2 and x_4 are in the same class, and that x_3, x_5, and x_6 are in the same class. Hence if this term is inserted in an empty tree, we obtain a tree with one node containing the substitution:

$x_1 = f(x_2, x_3, x_4, x_5, x_6),\;\; x_2 = x_4,\;\; x_3 = x_5,\;\; x_3 = x_6$

(or with other, equivalent but always non-redundant, equality constraints).

The insertion process in a non-empty tree first searches for the node where insertion will take place. This search process is like matching, except for two aspects. Firstly, the external variables of the index are only allowed to be instantiated with variables of the inserted term. But since a variable x in the tree sometimes plays the role of an internal and external variable at the same time (see Example 9), one cannot know in advance which situation applies until a leaf is reached: if x has no occurrence as the left hand side of a substitution (not an equality constraint) along the path to the leaf, then it plays the role of an external variable for this leaf. Secondly, during insertion the equality constraints are checked on the associated information about the variables classes, instead of checking the syntactic equality of subterms.

If a siblings list is reached where no sibling has a total agreement with the inserted term then two different situations can occur. If there is a sibling with a partial agreement with the inserted term, then one takes the first such sibling (first-fit, this is what we do) or the sibling with the maximal (in some sense) agreement (best-fit). The substitution of this node is replaced with the common part (including the common equality constraints), and two new nodes are created with the remaining substitutions. If a point is reached where all sibling nodes have an empty common part with the inserted substitution, then the inserted substitution is added to the siblings list. In both situations, the remaining substitution of the inserted term is built including the equality relations that have not been covered by the equality constraints encountered along the path from the root.

Deletion is also tricky, mainly because finding the term to be deleted requires again to control the equality constraints and the instantiation of external variables only with variables of the term to be found.

It is important to be aware of the fact that updates cost little time in practice, because updates are relatively unfrequent compared with retrieval operations. Experiments seem to confirm that there are usually less than one update per hundred matching retrievals.

5. Coded Context Trees

One of the conclusions that can be drawn from the experiments of (Nieuwenhuis et al., 2001) is that it does in fact pay off to compile an index into a form of interpreted abstract instructions as suggested by the code trees method of (Voronkov, 1995) (similar findings have also been obtained in the field of logic programming).

5.1. Advantages of coding

For context trees, consider for example again the `SubstMatch` loop we saw before. Instead of a linked list of substitution pairs to be traversed, one can simply use a piece of code formed by contiguous abstract code instructions, one for each substitution pair. One important advantage of this approach is that no control (like the outermost `if` statement of `SubstMatch`) has to be looked up.

In addition, one can use instructions decomposing operations like `TermMatch` into sequences of instructions for the concrete second argument, which is known at compile (i.e., index update) time. That is, instead of calling a general-purpose operation

`TermMatch(queryTerm,setTerm)`

one has instructions that do

`TermMatchWithThisSetTerm(queryTerm)`.
These advantages give more speedup than what has to be paid for in overhead arising from the need for interpreting the operation code of the abstract instructions. The latter is just a `switch` statement that in modern compilers becomes a constant time computed-jump instruction.

5.2. Registers

Consider in Fig. 1 the node that is the second child of the root. It contains the substitution pair $2 = ((f4)5)$. During a retrieval operation visiting this node, variable 4 will get instantiated with some subterm t of the query. This subterm t can be inspected at several places in the subtree below this node (each time variable 4 is referred to).

An important question that arises is where to store this pointer to t. One possibility is to implement the references to variable 4 in the subtree as pointers to the actual variable argument of the term $((f4)5)$, which is the one that points to t. However, this is not very convenient because it produces too many cache faults, since in larger trees the accesses to variable 4 can be far below this node. Therefore, a better idea is to have special registers for the variables, and have variable 4 mean register no. 4. As we will see, this also saves space.

Now we address the problem of how to keep the number of registers small. It is not difficult to see that the same register can be re-used on independent paths in the context tree, and that on each path in the tree, at most as many registers are needed as the size of the term stored along this path. However, most insertions require the replacement of an internal node of the tree by a common part node and two rest nodes (as explained in Section 4.4). In this common part node new internal variables are created, and we have found no suitable way for deciding which registers should be assigned to these new variables (without inspecting which registers are used in the whole subtree below).

Therefore we decided to make registers correspond to positions of the terms stored: the root register is register 1, and the left (right) child position of a position p is $2p$ (resp. $2p + 1$). For example, if in a node we have a substitution like $l = ((fr_1)r_2)$ then $r_1 = 4l$ and $r_2 = 2l + 1$.

In our implementation we want register numbers to be stored in two bytes. Therefore we have $2^{16} \approx 64000$ registers. We keep half of them (i.e., 2^{15}) for all positions up to depth 15 in the terms. The other half is kept for all deeper postions, and is assigned by hashing on the positions.

5.3. The Code

We have instructions for the most frequent substitution pairs (other pairs can be expressed by combinations of these). For example, one of the instructions handled in the part of our abstract machine interpreter that is given below is the abstract instruction called `lcvv`. This instruction corresponds to a substitution pair $left = ((cv)v)$ where c is a constant and both v's are variables. This instruction needs 8 bytes: 1 for its operation code `opcode`, 1 for the constant `c` 2 for its left hand side register `left`, and 4 for the instruction `failInst` to jump to in case of failure. For applications where more than 256 different constants might be needed (e.g., where new constants are introduced dynamically), one could of course allocate more space for each constant.

For each instruction the abstract machine checks whether there is failure or not. In case of success, always the next instruction to be executed is immediately below. Hence all such sequences of contiguous instructions end with a `success` instruction that returns the number of the term stored at the corresponding leaf of the context tree (this number is typically used for recovering the information associated with the term). The instructions for which there is no `failInst` to jump to in case of failure (because they have no node in the tree to backtrack to) have a `failInst` field that points to a fake `success` instruction returning term number 0. Note that altogether there is absolutely no overhead due to any control lookup.

There are no other pointer structures than the `failInst` fields, i.e., the context tree only consists of the code. Two additional bits (taken from the `opcode` field) suffice for recovering the exact tree structure at update time, by indicating whether this instruction is the start of a new child, or whether its node is the last one of a list of siblings. Hence the only memory used is the one of the code itself, plus the static memory of the register array.

The other two instructions dealt with in the part of our abstract machine interpreter that is given below are `c`, for substitution pairs $left = c$ where c is a constant, and `eq`, for equality constraints between two registers `r1` and `r2`. These two instruction occupy 8 and 12 bytes, respectively.

The variable `pc` contains the current program counter of our abstract machine, and the abstract machine simply consists of a switch statement:

```
switch (pc->opcode)
  case c:
     s = registers[pc->left];
     if ( s != pc->c ) {pc = pc->failInst; goto switch;}
```

```
    pc = pc+8;  goto switch;
  case lcvv:
    s = registers[pc->left];
    if ( TermIsConstant(s)           ||
         TermIsConstant(LeftChild(s)) ||
         LeftChild(LeftChild(s)) != pc->c )
                {pc = pc->failInst; goto switch;}
    registers[4*pc->left+1] = RightChild(LeftChild(s));
    registers[2*pc->left+1] = RightChild(s);
    pc = pc+8;  goto switch;
 case eq:
  if ( !TermEqual(registers[pc->r1],registers[pc->r2]) )
                {pc = pc->failInst; goto switch;}
    pc = pc+12; goto switch;
 ...
```

In our current implementation there are 14 instructions. For instance, other instructions are `vv` and `cv` (i.e., for substitution pairs of the form $l = (vv)$ and $l = (cv)$, respectively). For instructions like `lcvv` there are also special versions that deal with deep registers (i.e., registers at positions deeper than 15). Note that in those special versions one cannot compute the numbers of the registers at the right (like the computation `4*pc->left+1` in the `lcvv` instruction), because these registers are assigned by hashing instead of according to positions. Therefore in those special instructions (which are scarce in real examples) the registers at the right are also stored in the instruction itself. The special version of `lcvv` then needs 12 bytes instead of 8.

6. Experiments

In our experiments we have adopted the methodology described in (Nieuwenhuis et al., 2001). (In that paper one can find a detailed discussion of how to design experiments for the evaluation of indexing techniques so that they can be repeated and validated by others without difficulty.) For the purposes of this article, we ran large benchmarks, each containing the exact sequence of (thousands of update and millions of retrieval) operations on the matching index as they are executed when running one of three well-known state-of-the-art provers on certain problems drawn from various subsets of the TPTP problem date base (Sutcliffe et al., 1994).

Typically, (at least, for these benchmarks) almost all instructions in the tree are `c`, `vv`, `cv`, `eq`, `success`, and `lcvv`, in more or less equal proportions. The tree tends to have around 3 instructions per node, and around 2.5 children per node.

At retrieval time, the most frequently visited instructions are `c` (around 40%), `vv` and `cv` (around 20% each), and `eq` (around 15%).

It is also interesting to observe that `eq` instructions almost always fail (usually around 90% of the cases), and `c` also tends to fail in at least 70% of the cases. The instruction `vv` fails less: around 30-40% of the cases. On average, around 60% of the instructions fail. In general, matches usually only succeed in less than 10% of the cases. In most benchmarks, for one query an average of between 20 and 40 abstract instructions are executed, at a rate of around 30 million abstract instructions per second on a 1GHz Pentium-III laptop.

Comparisons have been made between our implementation (column "Cont." in Figure 6 below) with the discrimination tree implementation (column "Disc.") of the Waldmeister prover (Hillenbrand et al., 1997), and the code trees (column "Code") of the Vampire prover (Voronkov, 1995), as provided by their own implementors. Code trees are, conceptually, a refined form of standard discrimination trees. In their latest version (Riazanov and Voronkov, 2000), code trees apply a similar treatment of the equality tests as the one of (Nieuwenhuis et al., 1997) we use here. In the Figure 6 given below, the first column indicates the TPTP problem together with the prover which originated the benchmark (with F/V/W standing for the provers Fiesta, Vampire and Waldmeister, respectively).

Figure 6 shows that our implementation it is quite competitive in time. In fact, on these benchmarks it is even slightly faster (49.97 seconds total time) than the (more mature) implementation of the Vampire Code Trees (52.93 seconds total time). These times include the update operations, which in our current implementation (for which we have not concentrated yet on optimizing updates) in some examples take up to 20% of the time. We believe that this can be brought down to between 5 and 10%, like it happens in Vampire's Code Trees.

Moreover, context trees are, as expected, by far best in space, due tp the higher degree of sharing. Therefore, it will be interesting to do experiments with benchmarks where the index becomes really large. The authors suspect that, due to their lower memory consumption, context trees will scale relatively better compared with the other indexing techniques.

7. Conclusions and future work

The concept of context trees has been introduced and we have shown (and experimentally verified) that large space savings are possible. We have described in detail how these trees can be efficiently implemented. From the performance of our current implementation it can be seen that context trees have a great potential for applications in automated theorem proving. Due to the high degree of sharing, they allow for efficient matching, they require much less memory, and yet the time needed for the somewhat more complex updates remains negligible.

One line of future research in context trees concerns other retrieval operations. In particular, we are currently designing context trees for backward matching.

One the other hand, concerning the forward matching retrieval studied in this article, there is at least one direction for further work from which further substantial improvements could be obtained: the exact computation of backtracking nodes. Far more information than we have discussed so far can be precomputed at update time on the index. We describe one of the more promising ideas that should help to considerably reduce the amount of nodes visited at retrieval time.

Consider an occurrence p of a substitution pair $x_i = t$ in a substitution of the tree. Denote by $accum(p)$ the term that is, roughly speaking, the accumulated substitution from the root to the pair p, including p itself. If, during matching, a failure occurs just after the pair p, then the query term is an instance of $accum(p)$ (and this is the most general statement one can make at that point for all possible queries). This knowledge can be exploited to exactly determine the node to which one should backtrack. Let p' be first pair after p in preorder traversal of the tree whose associated term $accum(p')$ is unifiable with $accum(p)$. Then $accum(p')$ is the "next" term in the tree that can have a common instance with $accum(p)$. Therefore, $accum(p')$ is precisely the next term in the tree of which the query can be an instance as well! Hence the backtracking node to which one should jump in this situation is the one just after p'.

It seems possible to recompute locally, upon each update of the tree, the backtracking pointers associated to each substitution pair, and store these pointers at the pair itself, thus actually minimizing (in the strictest sense of the word) the search during matching. We are currently working out this idea in more detail.

References

Christian, J.: 1993, ‘Flatterms, Discrimination Nets, and Fast Term rewriting’. *Journal of Automated Reasoning* **10**, 95–113.

Graf, P.: 1995, ‘Substitution Tree Indexing’. In: J. Hsiang (ed.): *6th RTA*. Kaiserslautern, Germany, pp. 117–131, Springer-Verlag.

Hillenbrand, T., A. Buch, R. Vogt, and B. Löchner: 1997, ‘WALDMEISTER—High-Performance Equational Deduction’. *Journal of Automated Reasoning* **18**(2), 265–270.

McCune, W.: 1992, ‘Experiments with discrimination tree indexing and path indexing for term retrieval’. *Journal of Automated Reasoning* **9**(2), 147–167.

Nieuwenhuis, R., T. HillenBrand, A. Riazanov, and A. Voronkov: 2001, ‘On the evaluation of indexing techniques for theorem proving’. In: L. Goré and Nipkow (eds.): *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2001)*, Vol. 2038 of *LNAI*. Siena, Italy, pp. 257–271, Springer-Verlag.

Nieuwenhuis, R., J. M. Rivero, and M. Vallejo: 1997, ‘A Kernel of Data Structures and Algorithms for Automated Deduction with Equality Clauses (System description)’. In: W. McCune (ed.): *14th International Conference on Automated Deduction (CADE)*. Jamestown, Australia, pp. 49–53, Springer-Verlag. Long version at www.lsi.upc.es/˜roberto.

Riazanov, A. and A. Voronkov: 2000, ‘Partially Adaptive Code Trees’. In: *Proceedings of JELIA 2000*. Malaga, Spain.

Sutcliffe, G., C. Suttner, and T. Yemenis: 1994, ‘The TPTP problem library’. In: A. Bundy (ed.): *Proceedings of the 12th International Conference on Automated Deduction*, Vol. 814 of *LNAI*. Berlin, pp. 252–266, Springer.

Voronkov, A.: 1995, ‘The Anatomy of Vampire. Implementing Bottom-up Procedures with Code Trees’. *Journal of Automated Reasoning* **15**(2), 237–265.

Weidenbach, C.: 1997, ‘SPASS—Version 0.49’. *Journal of Automated Reasoning* **18**(2), 247–252.

benchmark from prover	time in seconds Code	Disc.	Cont.	space in KB Code	Disc.	Cont.
F COL002-5	0.45	0.72	0.45	892	10658	310
F COL004-3	0.36	0.65	0.29	77	5647	27
F GRP164-1	6.46	11.01	6.36	5623	33610	1750
F GRP179-2	6.50	10.93	6.43	5398	33752	1705
F LAT023-1	0.39	0.64	0.34	188	6214	72
F LAT026-1	0.38	0.61	0.34	354	7376	125
F LCL109-2	0.19	0.26	0.17	492	6849	147
F RNG020-6	0.86	1.51	1.08	528	7359	170
F ROB022-1	0.30	0.52	0.38	113	5998	37
V CAT001-4	1.08	2.42	1.42	3791	18710	1033
V CAT002-3	0.98	2.23	1.26	2439	13856	650
V CAT003-4	1.08	2.39	1.45	3760	18167	1026
V CIV003-1	2.28	3.04	3.08	3563	27230	1006
V COL079-2	1.96	3.17	1.34	2730	14078	752
V HEN011-2	1.14	1.48	0.89	206	5937	65
V LAT002-1	2.07	2.99	1.98	3066	19175	798
V LCL109-4	1.94	3.16	2.66	6551	28975	1632
V RNG034-1	1.19	2.17	1.46	2498	13254	708
V SET015-4	0.93	1.34	0.88	305	5949	78
W BOO015-4	0.09	0.13	0.09	10	4783	4
W GRP024-5	1.36	2.16	1.40	18	5507	7
W GRP187-1	3.84	5.54	3.19	90	5467	31
W LAT009-1	1.36	2.13	0.90	18	5499	6
W LAT020-1	6.98	10.51	4.53	28	5891	10
W LCL109-2	0.19	0.28	0.15	15	5143	5
W RNG028-5	1.91	3.18	1.52	27	6231	9
W RNG035-7	3.31	5.67	3.24	34	5911	12
W ROB026-1	3.36	6.67	2.74	67	6423	23

Figure 2. Experimental results on a 1GHz Pentium-III laptop.