Context Trees[⋆]

Harald Ganzinger[1], Robert Nieuwenhuis[2], and Pilar Nivela[2]

[1] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany.
hg@mpi-sb.mpg.de
[2] Technical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain.
roberto@lsi.upc.es nivela@lsi.upc.es

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 describe an implementation of context trees based on Curry terms and on an extension of substitution trees with equality constraints, where one also does not distinguish between internal and external variables. Experiments with matching benchmarks show that our preliminary implementation is already 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.

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 generalization 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 much faster than memory capacity and one may assume that this gap will become even 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 dominant factor for

[⋆] The second and third author are partially supported by the Spanish CICYT project HEMOSS ref. TIC98-0949-C02-01. All test programs, implementations and benchmarks mentioned in this paper are available at www.lsi.upc.es/∼roberto.

time, instead of processor speed. Therefore, in what follows we will mainly focus on matching retrieval operations and on memory consumption.

One important aspect makes indexing techniques in theorem proving essentially different from indexing in other contexts like functional or logic programming: the index is subject to insertions and deletions. 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* [1, 4], the compiled variant of discrimination trees, called *code trees* [9], and *substitution trees* [2].

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 $f(h(b), g(y), h(y))$ will be stored in a substitution tree and discrimination tree, respectively, as shown:

substitution tree:

$$\begin{tikzcd}
 & x_0 = f(x_1, g(x_2), h(y)) \arrow[dl] \arrow[dr] & \\
x_1 = a,\, x_2 = x & & x_1 = h(b),\, x_2 = y
\end{tikzcd}$$

discrimination tree:

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

In a substitution tree all terms $x_0\sigma$ 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 $x_0 = f(a, g(x), h(y))$. When inserting the second term, internal variables are placed at the points of disagreement, and children are created with the "remaining" substitutions of both. Therefore all common contexts can be shared. □

Example 2. It clear that the additional sharing in substitution trees avoids repeated work (which is the main goal of 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. But, on the other hand, in substitution trees the basic traversal algorithms are significantly more costly. □

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:

$$\begin{tikzcd}
 & & x0 = h(x1, x2) \arrow[dl, dash] \arrow[dr, dash] & \\
 & x1 = x \arrow[dl, dash] \arrow[dr, dash] & & x1 = b,\ x2 = f(t) \\
x2 = f(t) & & x2 = g(t) &
\end{tikzcd}$$

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:

$$\begin{tikzcd}
 & & x0 = h(x1, F(t)) \arrow[dl, dash] \arrow[dr, dash] & \\
 & x1 = x \arrow[dl, dash] \arrow[dr, dash] & & x1 = b,\ F = f \\
F = f & & F = g &
\end{tikzcd}$$

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 some evidence for its practical value. First, we show how they can be adequately implemented. In order to be able to reuse some of the main ideas for efficient implementation of substitution trees, we will consider terms built from a single pairing constructor and constants. These terms will also be called *Curry terms*. We describe an implementation based on these Curry terms and an extension of substitution trees by equality constraints and by not distinguishing *internal* and *external* variables. The second evidence for its practical value is empirical. Experiments with matching show that our preliminary implementation (which does not yet include several important enhancements) is already competitive with tightly coded state-of-the-art implementations, namely the implementation of discrimination trees of the Waldmeister prover [3] and the code trees of the Vampire prover [9].

For the experiments, we adopted the methods for evaluation of indexing techniques described in [5]: (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 selected set of 10 problems; (ii) comparisons are made with the discrimination tree implementation of the Waldmeister prover [3], and the code trees of the Vampire prover [9], as provided by their own implementors using the test driver of [5].

This paper is structured as follows. Section 2 introduces some basic concepts of indexing, discrimination trees and substitution trees. In Section 3 we outline some problems with direct implementations of context trees and explain how one can use Curry terms to solve theses problems. We also show that the use of Curry terms has several additional advantages. In Section 4 we describe our implementation in a certain detail. Finally, Sections 5 and 6 describe the experimental results and 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 [1], 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 kind of nodes lead 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 form 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 principle; see also below), all matches found are correct and no later equality tests are needed. The Waldmeister prover [3] 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 [2], who also developed an implementation that is still used in the Spass prover [10]. (A more efficient implementation was given in the context of the Dedam (Deduction abstract machine) kernel of data structures [6], and has served as a basis for our implementation of context trees as well.)

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:

$$\begin{array}{r|c|c|}
\cline{2-3}
a & f & \\
\cline{2-3}
a+1 & \texttt{ref} & a_1 \\
\cline{2-3}
 & \vdots & \vdots \\
\cline{2-3}
a+n & \texttt{ref} & a_n \\
\cline{2-3}
\end{array}$$

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 are always 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_0 at the root with the query term, and traverses the tree, where at each visited node with a substitution $x_1 = t_1, \ldots, x_n = t_n$, one performs the basic term-to-term operation (syntactic equality, matching, unification) between each (already instantiated) x_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.

Due to space limitations, we cannot go into details here about the update operations for substitution trees. Let us only mention that several insertion strategies are possible (first-fit, best-fit), and that the basic operation for insertion is the computation of the *common* part and *remainders* of two substitutions. For deletions, one sometimes needs to apply the reverse operation, namely to *merge* two substitutions into a single one.

Example 4. Let σ_1 be the substitution $\{x_1 = g(a, h(b)),\ x_2 = a\}$ and let σ_2 be $\{x_1 = g(b, h(c)),\ x_2 = b\}$. Their common part is $\{x_1 = g(x_3, h(x_4))\}$. The remainders of both substitutions are $\{x_2 = a,\ x_3 = a,\ x_4 = b\}$ and $\{x_2 = b,\ x_3 = b,\ x_4 = c\}$ respectively. □

2.2 Substitution trees for matching

In Dedam 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 5. 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 $x_0 = f(x_1, x)$, with children $x_1 = x$ and $x_1 = a$. When matching, at the root x_1 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. □

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. 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 the *equality constraints* at the leaves.

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 6. 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 remaining parts are $\{F = f\}$ and $\{F = g,\ x_j = s\}$, respectively, and we obtain:

$$\begin{tikzcd}[column sep=small]
x_i = f(x_j,t) \arrow[d, dash] & & & x_i = F(x_j,t,u) \arrow[dl, dash] \arrow[dr, dash] & \\
T & & F = f \arrow[d, dash] & & F = g,\ x_j = s \\
 & & T & &
\end{tikzcd}$$

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 7. 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 specialized substitution trees for matching requires to deal with the specific properties of the context variables.

Example 8. Consider again Example 6. 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 previous problem 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 @, 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 9. Consider again the terms of Example 6, where we saw that one can share the term t in $f(x_j, t)$ and $g(s, t, u)$ by having $F(x_j, t, u)$. In Curry form, these terms become $@(@(f, x_j), t)$ and $@(@(@(g, s), t), u)$ 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 $@(@(f, t), x_j)$ and $@(@(@(g, u), t), s)$. 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 $@(@(x_k, t), x_j)$, where the remaining parts are $\{x_k = f\}$ and $\{x_k = @(g, u),\ x_j = s\}$.

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, since it is obviously not necessary to store any apply symbols, all memory blocks contain exactly two heap cells.

Example 10. The term $f(b, g(x))$ becomes $@(@(f, b), @(g, x))$, which can be written in pair notation simply as $((f, b), (g, x))$. Compare the Prolog format with how the Curry term can be stored:

```
  Prolog term:               Curry Term:


10 f                       100 ref -->  120 ref -----------> 150 f
11 ref --> 40 b                         121 ref --> 140 g    151 b
12 ref -------> 60 g                                141 var
                61 ref --> 80 ref
```

Note that in the Prolog term, constants are a block of heap cells on their own such as the b at address 40. Alternatively, constants can also be placed directly at pointer position for minimizing space. But with Prolog terms this makes the algorithms slower as a uniform treatment for all function symbols (constants or not) becomes impossible. But in Curry terms, since *all* function symbols are constants, this space optimization can be used without any cost. □

In Curry terms each cell is either a constant, a variable `var` or a `ref` to a subterm. Curry terms are always headed by a single heap cell, and all other blocks consist of two contiguous heap cells. This makes the basic algorithms very efficient, as exemplified by the following recursive algorithm for testing the equality of two (ground) Curry terms:

```
int TermEqual(_HeapAddr s, _HeapAddr t){
    if (HeapTag(s)!=HeapTag(t)) return(0);
    if (HeapIsRef(s)){
        if (!TermEqual(HeapAddr(s),  HeapAddr(t)  )) return(0);
        if (!TermEqual(HeapAddr(s)+1,HeapAddr(t)+1)) return(0);}
    return(1);}
```

4 Implementation

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 (see [6]). 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:

$$\begin{tikzcd}
 & & x_0 = f(x_1, x, y, z, a),\ x = y \arrow[dl, no head] \arrow[dr, no head] & \\
 & x = z \arrow[dl, no head] \arrow[dr, no head] & & x_1 = c \\
x_1 = a & & x_1 = b &
\end{tikzcd}$$

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 MF-sets (merge-find sets).

4.2 Internal vs external variables

We have seen that in our Curry terms we only consider heap cells that are a constant, a `ref`, or a variable `var`. Indeed, it turns out that the usual distinction between internal and external variables can also be dropped. (A variable that is not instantiated 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 11. If the tree contains the two terms $f(a, a)$ and $f(x, b)$, we have:

distinguishing internal and external vars.:

$$\begin{tikzcd}
 & x_0 = f(x_1, x_2) \arrow[dl, no head] \arrow[dr, no head] & \\
x_1 = a, x_2 = a & & x_1 = x, x_2 = b
\end{tikzcd}$$

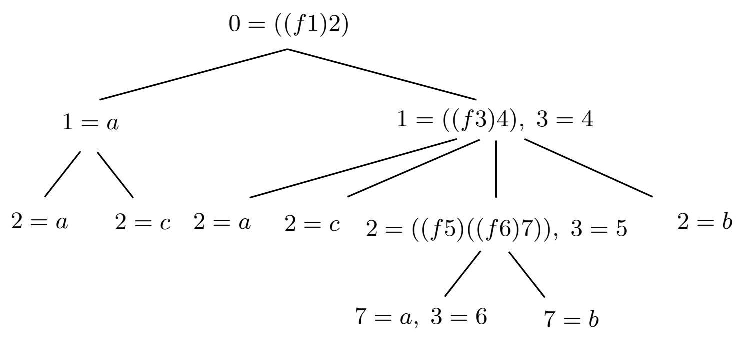
no distinctions:

$$\begin{tikzcd}
 & x_0 = f(x_1, x_2) \arrow[dl, no head] \arrow[dr, no head] & \\
x_1 = a, x_2 = a & & x_2 = b
\end{tikzcd}$$

Note that in the second tree the variable x_1 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 can be implemented as follows:

```
int TermMatch(_HeapAddr query, _HeapAddr set){
   if (HeapIsVar(set)) { HeapSetAddr(set,query); return(1); }
   if (HeapTag(query)!=HeapTag(set)) return(0);
   if (HeapIsRef(set)){
      if (!TermMatch(HeapAddr(query),  HeapAddr(set)  )) return(0);
      if (!TermMatch(HeapAddr(query)+1,HeapAddr(set)+1)) return(0);}
   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)	$f(0, b)$	$((f0)b)$

Fig. 1. Context tree for the terms 1)–7)

4.3 Matching retrieval

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 $3 = 4$ and $3 = 5$ are shared among several branches. Given the term-to-term operations for equality and matching from above, 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_0 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->firstChild ) return(1);
  if (!tree->nextSibling) return(CTreeMatch(tree->firstChild));
  return(CTreeMatch(tree->nextSibling)||CTreeMatch(tree->firstChild));}
```

This is the only retrieval algorithm that is not coded in our implementation as shown here. In the implementation, it is iterative and uses a backtracking stack. The other recursive algorithms for term-to-term equality and matching are also recursive in our current implementation, in the form we have given them above.

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 several MF-sets for keeping the information about the equivalence classes of the variables. For instance, the term $f(x, y, x, y, y)$ is inserted as $f(x_1, x_2, x_3, x_4, x_5)$ with the associated information that x_1 and x_3 are in the same class, and that x_2, x_4, and x_5 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:

$x_0 = f(x_1, x_2, x_3, x_4, x_5),\ \ x_1 = x_3,\ \ x_2 = x_4,\ \ x_2 = x_5$

(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 11), 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. Moreover, unlike what happens in insertion, backtracking is needed for finding.

It is important to be aware of the fact that updates cost little time in practice, because updates are relatively infrequent compared with retrieval operations.

Experiments seem to confirm that there are only one or two updates per thousand retrievals. On the benchmarks of [5] (see below) in none of the benchmarks updates took more than 5 percent of the time.

5 Experiments

In our experiments we have adopted the methodology described in [5]. (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 the present paper, we ran 30 very 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 [8]. Comparisons are made between our preliminary implementation (column "Cont." in Figure 5 below) with the discrimination tree implementation (column "Disc.") of the Waldmeister prover [3], and the code trees (column "Code") of the Vampire prover [9], as provided and run by their own implementors.

The figure 5 shows that, in spite of the fact that our implementation can be much further improved (see Section 6), it is already quite competitive in time. Moreover, context trees are, as expected, best in space, except for the very small indexes (mostly coming from the Waldmeister prover). (A substantial further space improvement can be expected from a compiled implementation as sketched in section 6.2.) Code trees are, conceptually, a refined form of standard discrimination trees. In their latest version [7], code trees apply a similar treatment of the equality tests as the one of [6] we use here. The faster speed of code trees is, in our opinion, by and large due to the compilation of the index (see Section 6). We do not include here the results of the aforementioned Spass and Dedam implementations of substitution trees, because, with a similar degree of refinement of coding as our current implementation of context trees, they are at least a factor three slower and need much more space.

6 Conclusions and future work

The concept of context trees has been introduced and we have shown (and experimentally verified) that large space savings can be possible compared with substitution trees and discrimination trees. We have described in detail how these trees can be efficiently implemented. By representing terms in Curry form, an implementation can be based on a simplified variant of substitution trees. Already from the performance of our first (unfinished) 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.

TPTP problem	benchmark from prover	**time in seconds** Code	Disc.	Cont.	**space in KB** Code	Disc.	Cont.
COL002-5	Fiesta	1.30	1.55	2.61	925	6090	922
COL004-3	Fiesta	0.96	1.22	2.39	80	727	86
LAT023-1	Fiesta	1.10	1.49	1.92	198	1646	210
LAT026-1	Fiesta	1.11	1.49	1.79	373	2813	371
LCL109-2	Fiesta	0.47	0.65	0.80	508	2285	466
RNG020-6	Fiesta	2.26	3.19	5.33	544	2435	517
ROB022-1	Fiesta	0.92	1.20	2.23	119	1086	101
GRP164-1	Fiesta	17.60	24.25	32.40	5823	28682	5352
GRP179-2	Fiesta	18.34	24.25	32.40	5597	29181	5207
GRP196-1	Fiesta	6.96	11.92	15.45	1	543	1
BOO015-4	Waldmeister	0.25	0.31	0.46	11	575	11
GRP024-5	Waldmeister	3.54	4.82	7.44	19	591	22
GRP187-1	Waldmeister	10.44	11.68	17.64	96	903	97
LAT009-1	Waldmeister	3.78	4.97	5.97	19	591	20
LAT020-1	Waldmeister	17.74	24.97	29.87	30	631	31
LCL109-2	Waldmeister	0.49	0.66	0.82	16	591	15
RNG028-5	Waldmeister	4.19	6.66	9.08	28	607	29
RNG035-7	Waldmeister	8.20	12.10	18.55	36	647	37
ROB006-2	Waldmeister	9.88	14.31	21.60	128	1142	116
ROB026-1	Waldmeister	8.52	13.55	17.34	69	807	68
COL079-2	Vampire	5.46	8.41	7.24	2769	9158	2138
LAT002-1	Vampire	5.83	7.72	9.48	3164	14603	2554
LCL109-4	Vampire	5.62	7.65	13.02	6703	24403	4986
CIV003-1	Vampire	7.57	7.13	15.57	3754	22664	3081
RNG034-1	Vampire	3.27	4.95	6.86	2545	8330	2125
SET015-4	Vampire	2.54	2.69	4.53	314	1373	258
HEN011-2	Vampire	3.36	3.39	5.18	221	2069	211
CAT001-4	Vampire	3.28	5.74	7.11	3859	13786	3109
CAT002-3	Vampire	2.90	5.51	6.67	2483	9281	2021
CAT003-4	Vampire	3.21	5.82	6.90	3826	13595	3086

Fig. 2. Experimental results

With respect to our implementation, more work remains to be done regarding a tighter coding of the four (two of them recursive) algorithms used for retrieval — those we have seen in this paper. Experience with other implementations of term indexes has shown that this can give substantial factors of speedup.

Apart from these low-level aspects we also believe that there are at least two other directions for further work from which further substantial improvements will be obtained. We are going to describe them briefly now.

6.1 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 and to eliminate the need of a backtracking stack.

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.

6.2 Compiled context trees

One of the conclusions that can be drawn from the experiments of [5] 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 [9] (similar findings have also been obtained in the field of logic programming). For context trees, consider for example again the `SubstMatch` loop we saw before. Instead of such a loop, one can simply use a linked list of abstract code instructions like `TermEqual(adress1,addres2,FailureAddress)` where `FailureAddress` is the address to jump to in case of failure. The main advantage of this approach is that no control (like the outermost `if` statement of `SubstMatch`) has to be looked up, and, since the correct `address` arguments are already part of the abstract code, no instructions like `subst = subst->next` are needed and many indirect accesses like `subst->lhs` can be avoided.

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. Assume we specialized the `TermMatch` function for matching with the index term $(f(gx))$, that is,

```
10 ref -->  20 f
            21 ref   --> 30 g
                         31 var
```

This would give code such as the following sequence of 7 one-argument instructions:

```
if (!HeapIsRef(query))      goto fail;
query = HeapAddr(query);
if (HeapTag(query)!='f')    goto fail;
if (!HeapIsRef(query+1))   goto fail;
query = HeapAddr(query+1);
if (HeapTag(query)!='g')   goto fail;
HeapSetAddr(31,query+1)
```

By simple instruction counting, this code is easily shown to be far more efficient on an average query term than the general-purpose two-argument `TermMatch` function with $(f(gx))$ as second argument. All 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 all modern compilers, produces constant time code.

References

[1] Jim Christian. Flatterms, Discrimination Nets, and Fast Term rewriting. *Journal of Automated Reasoning*, 10:95–113, 1993.

[2] Peter Graf. Substitution Tree Indexing. In J. Hsiang, editor, *6th RTA*, LNCS 914, pages 117–131, Kaiserslautern, Germany, April 4–7, 1995. Springer-Verlag.

[3] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. WALDMEISTER—high-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, April 1997.

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

[5] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing data structures. 2001. This proceedings.

[6] Robert Nieuwenhuis, José Miguel Rivero, and Miguel Ángel Vallejo. A kernel of data structures and algorithms for automated deduction with equality clauses (system description). In William McCune, editor, *14th International Conference on Automated Deduction (CADE)*, LNAI 1249, pages 49–53, Jamestown, Australia, 1997. Springer-Verlag. Long version at www.lsi.upc.es/˜roberto.

[7] Alexandre Riazanov and Andrei Voronkov. Partially adaptive code trees. In *Proceedings of JELIA 2000*, Malaga, Spain, 2000.

[8] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 252–266, Berlin, June/July 1994. Springer.

[9] Andrei Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, October 1995.

[10] Christoph Weidenbach. SPASS—version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, April 1997.