# Datastructures for Resolution

Hans de Nivelle

Max Planck Institut für Informatik,
Im Stadwald
66123 Saarbrücken, Germany,
Email: nivelle@mpi-sb.mpg.de

**Abstract.** We present benchmark tests made when implementing the theorem prover Bliksem. We compared 5 different ways of implementing terms and atoms and 3 ways of implementing substitutions, based on deep and shallow binding. We also compared 3 different ways of implementing discrimination trees. In the last section we describe the implementation of forward subsumption used in Bliksem.

## 1 Introduction

We compare different implementation techniques for some of the datastructures that play a role in resolution. We implemented unification and matching for terms, different types of substitution for unification, and discrimination trees for retrieval of generalizations. For term represention we can conclude that it does not matter very much which implementation one chooses. There is only one exception: One must not choose the standard method, which is to represent a term $f(t_1, \ldots, t_n)$ as a record containing $f$ and a linked list containing the $t_i$. This method sometimes performs significantly worse than the other methods.

For substitutions we compared *deep* and *shallow* binding. (See [Bb74]) Deep substitutions are lists of pairs $(V, t)$, where $V$ is a variable, and $t$ is a reference to a term. In shallow substitutions the variables are represented by small integers and the substitution is represented as an array of values, indexed by the variables. We conclude that there is no clear winner here, but shallow substitutions have to be initialized. If one does this before every unification attempt, the initialization times dominate the unification times.

For discrimination trees we also distinguish deep and shallow trees. If one has only ground terms in the tree, than one should choose shallow trees. If there are also variables in the tree, then the times needed to control the variables become more important. The datastructure then becomes less important.

We think that it is useful to make such comparisons, even when the outcomes reveal no spectacular differences. Probably the main objection against papers like this is the fact that technical optimization is not worth the effort, or more precisely the fact that much more can be obtained by increased theoretical understanding. However technical optimization does not exclude theoretical improvement. The real problem is that too much a posteriori technical optimizations can make the program too large, and to rigid. We are not studying this type

of optimizations. We are studying a priori choices that have no consequences for the complexity of the program.

Another controversion is whether or not theorem provers should be tuned towards certain applications. Our opinion is that this can be done, but one should not claim that certain finite sets represent infinite application domains.

We conclude the paper with a short description of the forward subsumption check in Bliksem. This part is theoretical. We give an algorithm that has the same complexity as the algorithm in [GL85], but which does not compute connected components in advance.

## 2 Datastructures for Terms and Literals

Probably the most far reaching design decision is how atoms, terms and literals are implemented. The problems are caused by the fact that the length of a term is not known in advance, and by the internal tree-structure of terms. It appears to be the most natural solution to use techniques from Prolog. This is done in Spass [Spass99] and in Otter [McC99]. However in Waldmeister other datastructures are used. [Wm95].

We have implemented 5 ways of representing terms, and compared them on two fundamental operations, matching and unification in **unify.c**. Matching is one-sided unification, i.e. the problem of finding a substitution $\Theta$, such that $A\Theta = B$. This is used for simplification and the subsumption check. We first explain the different implementation techniques using the term $f(s(X), Y, s(a))$.

### 2.1 Pure Prefix, Prefix with Ends, and Flatterms

We begin with 3 essentially linear representations. The prefix representation is standard, flatterms were introduced by Christian in [Ch93]. The prefix with end representation appears to have been introduced by the author.

In the *prefix* representation the term is written in memory from left to right, in essentially the same manner as it would be written on paper without parentheses. The structure of the term is known, because each operator has a fixed arity. Traversing the term from left to right is easy because the term is stored in consecutive addresses in memory. However it is a problem to know where the term ends. In order to know this, additional administration is necessary. All algorithms (unification, substitution, copying, equality tests) that operate on terms need this administration. The same administration is necessary if one wants to access a subterm. In order to reach the $n$-th subterm one has to skip the first $n-1$ subterms, which needs administration. Nevertheless this method also has advantages because it uses little memory. It is argued in [Wm95] that there is intrinsic advantage in a short representation. It will turn out that the prefix representation does not perform worse than the deep term with argument lists representation, which uses 3 times more memory. The *prefix with ends* representation is similar to the prefix representation, but to each term a reference

to its end is added. This removes the need for the end-administration by the algorithms, which was the main disadvantage of the prefix representation. In this way both left/right traversal, and access of subterms is easy. The memory use is twice more than the memory use of pure prefix terms. The *Flatterm* representation was introduced by Christian. The flatterm representation is very similar to the prefix with end representation. Instead of storing the term on consecutive addresses in memory, a linked list is used. The following tables give representations of the term $f(s(X), Y, s(a))$. We write pointers as integers denoting the address of the object that they point to. In each case the term starts at location 100. For the flatterm representation the physical order in memory does not matter. We have chosen to put the term on consecutive addresses.

| Prefix | |
|---|---|
| address: | term |
| 100 : | $f$ |
| 101 : | $s$ |
| 102 : | $X$ |
| 103 : | $Y$ |
| 104 : | $s$ |
| 105 : | $a$ |

| Prefix with Ends | | |
|---|---|---|
| address: | term | end |
| 100 : | $f$ | 106 |
| 101 : | $s$ | 103 |
| 102 : | $X$ | 103 |
| 103 : | $Y$ | 104 |
| 104 : | $s$ | 106 |
| 105 : | $a$ | 106 |

| Flatterms | | | |
|---|---|---|---|
| address: | term | next | end |
| 100 : | $f$ | 101 | 106 |
| 101 : | $s$ | 102 | 103 |
| 102 : | $X$ | 103 | 103 |
| 103 : | $Y$ | 104 | 104 |
| 104 : | $s$ | 105 | 106 |
| 105 : | $a$ | 106 | 106 |

There is a representation missing, namely **listterms**. List terms are prefix terms, but using a list instead of consecutive addresses in memory. We do not expect good performance from this representation.

### 2.2 Deep Term Representations

In the *deep term* representation the tree-structure of the term is represented by references to the subterms. The deep term representation comes in two variations. One we call just the 'deep term' representation, the other we call the *deep term with argument list* representation, abbreviated as *deepwal*. These representations are the standard way used in logic or functional programming. In the deep term representation each term is stored as a record containing the main functor and references to the direct subterms of the term. If the functor has arity $n$, then the record has size $n + 1$. In the deepwal representation the subterms are stored in a linked list. This gives all records equal size. The deepwal representation is used in Spass and Otter. In both deep representations, accessing subterms is easy, but left/right traversal needs some additional administration. One can either use recursion, or a stack. The deep representation uses approximately twice more memory than the pure prefix representation, the deepwal representation uses approximately 3 times more memory. With both methods it is possible to *share subterms*. If the same subterm occurs more than once in memory, it is possible to use the same reference at all the places where it occurs. Checking this out is costly, but might be worth the effort. As is the case with flatterms, the term need not be on consecutive addresses. We kept the term ordered in the table to keep it readable. The term starts at location 100. In the

deep representation we use horizontal lines to separate the different records.

| Deep | |
|---|---|
| address: | term |
| 100 : | $f$ |
| 101 : | 120 |
| 102 : | 150 |
| 103 : | 170 |
| 120 : | $s$ |
| 121 : | 140 |
| 140 : | $X$ |
| 150 : | $Y$ |
| 170 : | $s$ |
| 171 : | 200 |
| 200 : | $a$ |

| Deepwal | | |
|---|---|---|
| address: | first | rest |
| 100 : | $f$ | 101 |
| 101 : | 110 | 102 |
| 102 : | 120 | 103 |
| 103 : | 130 | nil |
| 110 : | $s$ | 112 |
| 112 : | 115 | nil |
| 115 : | $X$ | nil |
| 120 : | $Y$ | nil |
| 130 : | $s$ | 131 |
| 131 : | 140 | nil |
| 140 : | $a$ | nil |

### 2.3 Reproducability of Measurements

We have implemented the unification and matching algorithm for the 5 datastructures, and collected running times for a collection of terms. Before we start doing experiments it might be a good idea to find a reliable manner of measuring. It turns out that this is not trivial. The following measurements are obtained using gprof and gcc. In the program **unify.c**, each of the 5 unification functions was called $10^6$ times. From this a profile was made using gprof. The numbers given in the table are the percentages of the total running time, as given by gprof. The numbers do not add up to 100% because gprof counts the time that it uses by itself.

| term1 | term2 | prefix | prefixwe | deep | deepwal | flat |
|---|---|---|---|---|---|---|
| $q(f(a,a),f(a,a))$ | $q(X,X)$ | 12.5 | 7.9 | 14.3 | 17.7 | 7.9 |
| | | 12.5 | 5.9 | 17.0 | 11.8 | 8.7 |
| | | 12.9 | 6.6 | 13.6 | 16.2 | 7.4 |
| | | 11.4 | 8.9 | 16.8 | 17.5 | 6.1 |
| | | 10.1 | 7.9 | 17.0 | 15.9 | 7.2 |
| | | 10.3 | 7.4 | 15.1 | 14.4 | 8.9 |
| | | 1.27 | 1.50 | 1.25 | 1.25 | 1.45 |

It can be concluded from this table that measurings made with gprof are pretty unreliable. The last row in the table contains the maximal value divided by the minimal value. The difference between the maximal and the minimal value can vary up to 50%. Moreover the ranking of the methods also varies.
The next table is obtained by using the time command. The time commands gives the total CPU time used by a process after it is finished. In each run only one of the unification functions is called $10^6$ times. So we had to make 5 runs

for each row in the table. The times are in seconds.

| term1 | term2 | prefix | prefixwe | deep | deepwal | flat |
|---|---|---|---|---|---|---|
| $q(f(a,a),f(a,a))$ | $q(X,X)$ | | | | | |
| | | 3.27 | 2.02 | 3.10 | 3.41 | 2.17 |
| | | 3.26 | 2.02 | 3.09 | 3.41 | 2.17 |
| | | 3.27 | 2.01 | 3.09 | 3.42 | 2.15 |
| | | 3.26 | 2.01 | 3.10 | 3.39 | 2.16 |
| | | 3.24 | 2.03 | 3.11 | 3.40 | 2.15 |
| | | 1.01 | 1.01 | 1.00 | 1.01 | 1.01 |

As can be seen from this table the behaviour of the time command is much better. The maximal time divided by the minimal time is nowhere greater than 1.01. Of course we do not know whether or not the values given here are in any relation with the real values, but at least they are consistent.

### 2.4 Measurements

The following table contains times for a various collection of terms, collected with **unify.c**. Some are unifiable, and some are not unifiable. The second contains times for matching obtained with **match.c**. The matching functions try to match the first term into the second term.

**Unification Times in $10^{-6}$ seconds**

| term1 | term2 | prefix | prefixwe | deep | deepwal | flat |
|---|---|---|---|---|---|---|
| $r(a,a,a)$ | $q(a,a)$ | 0.50 | 0.43 | 0.34 | 0.35 | 0.43 |
| $r(a,a,a)$ | $r(b,a,a)$ | 0.82 | 0.59 | 0.79 | 0.78 | 0.60 |
| $q(f(a,a),f(a,a))$ | $q(X,X)$ | 3.27 | 2.01 | 3.10 | 3.41 | 2.15 |
| $q(X,X)$ | $q(f(a,a),f(a,a))$ | 3.19 | 1.96 | 3.12 | 3.43 | 2.11 |
| $q(f(a,a),f(a,a))$ | $q(X,Y)$ | 3.13 | 2.10 | 3.47 | 3.69 | 2.26 |
| $q(f(a,a),f(a,b))$ | $q(f(a,a),f(a,b))$ | 2.61 | 1.41 | 2.74 | 3.17 | 1.51 |
| $q(f(a,a),f(a,b))$ | $q(f(a,a),f(a,a))$ | 2.36 | 1.37 | 2.71 | 3.00 | 1.46 |
| $q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(Y))))$ | $q(X,X)$ | 6.48 | 4.19 | 6.75 | 8.06 | 4.52 |
| $q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(X))))$ | $q(X,X)$ | 6.10 | 3.77 | 6.12 | 7.31 | 4.03 |
| $q(f(X,Y),f(X,Z))$ | $q(f(Y,X),f(Z,X))$ | 6.08 | 4.28 | 5.30 | 5.61 | 4.38 |
| $q(s(s(f(X,Y))),s(Y))$ | $q(X,s(s(s(X))))$ | 3.79 | 2.79 | 4.92 | 17.74 | 3.12 |

**Matching Times in $10^{-6}$ seconds**

| term1 | term2 | prefix | prefixwe | deep | deepwal | flat |
|---|---|---|---|---|---|---|
| $r(a,a,a)$ | $q(a,a)$ | 0.37 | 0.39 | 0.23 | 0.24 | 0.34 |
| $r(a,a,a)$ | $r(b,a,a)$ | 0.55 | 0.50 | 0.58 | 0.54 | 0.46 |
| $r(a,a,a)$ | $r(a,b,a)$ | 0.72 | 0.60 | 0.90 | 0.86 | 0.58 |
| $r(a,b,a)$ | $r(a,b,b)$ | 0.84 | 0.71 | 1.21 | 1.18 | 0.69 |
| $r(a,b,a)$ | $r(a,b,a)$ | 0.92 | 0.73 | 1.30 | 1.34 | 0.77 |
| $r(X,Y,Z)$ | $r(s(a),s(b),f(X,Y))$ | 2.22 | 1.30 | 1.78 | 1.87 | 1.34 |
| $r(X,Y,X)$ | $r(s(f(a,b)),s(f(b,b)),$ $\ldots s(s(f(a,b))))$ | 2.64 | 1.51 | 2.41 | 2.75 | 1.63 |
| $r(s(X),f(X,Y),f(Y,X))$ | $r(s(a),f(a,b),f(b,a))$ | 2.86 | 2.48 | 3.70 | 4.03 | 2.58 |
| $r(X,X,X)$ | $r(s^{10}(a),s^{10}(a),s^{10}(a))$ | 5.26 | 3.02 | 6.11 | 41.26 | 3.62 |

And the winners are ...
flatterms and prefix with end terms. It is not surprising that they perform approximately the same, because they are very similar. However it can be said that the differences are not essential. The only exception is that deepwal sometimes performs spectacularly worse than the other representations, see the last row of the table. Also interesting is that prefix performs better than deepwal, while it uses only one third of the memory. Despite of this, deepwal is used in many theorem provers.

## 3 Implementation of Substitutions

A *substitution* is formally defined as a finite set of assignments of the form $\{X_1 := t_1, \ldots, X_n := t_n\}$. The $X_i$ are distinct variables, and the $t_i$ are terms. In practice things are a bit more involved. It is convenient to see a substitution as a list $[X_1 := t_1, \ldots, X_n := t_n]$, where the assignments should be applied from left to right. This is the way in which substitutions are constructed by the unification algorithm. (For the matching algorithm it does not matter) It is also necessary to keep track of where variables come from. When resolvents/paramodulants are formally defined it is assumed that there are no overlapping variables between the parent clauses. It is not practical to implement resolution/paramodulation in this way, because a clause can paramodulate/resolve with itself. The following resolution derivation would fail without renaming in the occurs check:

$$\{\neg\, p(X,Y), p(s(Y),s(X))\} \Rightarrow \{\neg\, p(X,Y), p(s^2(X),s^2(Y))\}.$$

One cannot keep two copies of the same clause (if one has hyperresolution the situation is even worse) and it is inefficient to construct a renaming in advance, when construction of a resolvent is attempted. There are two practical solutions: The first is to construct two substitutions, one for each copy of each parent clause. The other is to construct one substitution, but to store both the variable and the parent clause (an integer identifying the copy) in the substitution. In either way it has to be stored for each right hand side of the substitution from which copy the term originates. Consider the following clause $\{\neg\, p(X,a), p(s(X),X)\}$

which can resolve with itself. We call the copy in which $\neg\, p(X, a)$ is used 1 and the copy in which $p(s(X), X)$ is used 2. The two alternatives then become:

$$\Theta = [X : 1 \ := s(X) : 2, \quad X : 1 \ := a : 1],$$

and

$$\Theta_1 = [X := \ s(X) : 2], \quad \Theta_2 = [X := \ a : 1].$$

One need not store numbers in the substitutions, one could also use references to the substitution that should be applied on the variables in the term. As is evident from this example, the difference between the two methods of storing substitutions is small. There is also no significant difference in the experiments. In the table the first method is called *deep1*, and the second method is called *deep2*.

A more important issue is the distinction between *deep* and *shallow* binding. With **deep binding** the substitutions are stored in an array or list, as pairs $(X, t)$. An assignment is added by adding it at the end of the list. An assignment is looked up by searching the list. An assignment can be deleted by removing it from the end of the list. Removing an assignment in the middle would be harder, but assignments are always added and removed in a stack like fashion. With **shallow binding** the fact is used that variables are (or can be represented as) small integers. The integers are used to index an array, in which the assignments are stored. Let $\Theta$ be the array. Then the assigment $X := t$ can added by assigning $\Theta[X] := t$. The value of $X$ can be found as $\Theta[X]$. An assignment is removed by assigning $\Theta[X] := \perp$. Here $\perp$ is some object different form all terms. Assignments can be added, removed and retrieved in constant time. The big disadvantage is that the array has to be initialized before use, by assigning $\Theta[V] := \perp$ for each possible variable $V$. Here one has to use some value higher than all possible variables. (One could register the number of variables in each clause. Then it is known in advance up to how far the array $\Theta$ has to be initialized) Shallow bindings are used in Otter and in Spass, with initializations up to 50. The initialization is very expensive, as can be seen from the tests below.

Another disadvantage of shallow binding is that it is not well adapted to backtracking. During backtracking assignments are added and removed in a stack like fashion, for example when searching in a discrimination/substitution tree, or when searching for hyperresolvents. With shallow binding one has to keep track of the order in which the assignments were made, so that they can be removed in the order in which they were added. This aspect plays no role in the tests that we made.

We made experiments for the two ways of deep binding, and for shallow binding. The results of the experiments are given below. In the column for shallow2, the first value is obtained by initializing up to 5. The second value is obtained by initializing up to 20. In Otter and Spass shallow binding with initialization up to 50 is used.

**Unification Times in $10^{-6}$ seconds**.

| term1 | term2 | deep1 | deep2 | shallow2 / 20 |
|---|---|---|---|---|
| $r(a,a,a)$ | $q(a,a)$ | 0.45 | 0.46 | 0.93/2.91 |
| $r(a,a,a)$ | $r(b,a,a)$ | 0.58 | 0.63 | 1.10/3.19 |
| $q(f(a,a),f(a,a))$ | $q(X,X)$ | 1.99 | 2.10 | 2.52/4.30 |
| $q(X,X)$ | $q(f(a,a),f(a,a))$ | 1.96 | 2.09 | 2.53/4.31 |
| $q(f(a,a),f(a,a))$ | $q(X,Y)$ | 2.11 | 2.02 | 2.17/3.97 |
| $q(f(a,a),f(a,b))$ | $q(f(a,a),f(a,b))$ | 1.39 | 1.43 | 2.02/3.88 |
| $q(f(a,a),f(a,b))$ | $q(f(a,a),f(a,a))$ | 1.37 | 1.40 | 1.99/3.86 |
| $q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(Y))))$ | $q(X,X)$ | 4.18 | 4.07 | 3.92/5.74 |
| $q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(X))))$ | $q(X,X)$ | 3.79 | 3.64 | 3.88/5.69 |
| $q(f(X,Y),f(X,Z))$ | $q(f(Y,X),f(Z,X))$ | 4.30 | 3.62 | 3.33/5.14 |

We suspect that the observed success of FPA's can be explained from the long initialization times. [Gr94b].

## 4 Discrimination Trees

Discrimination trees are obtained if one represents terms/literals in prefix notation and merges common prefixes as much as possible. Discrimination trees can be used when searching for matching terms, matched terms, or unifiers. At present we have experimental results only for the retrieval of matching terms. Profiles made on runs of Bliksem indicate that this is the most time critical. We have done theoretical work on unification in the past ([dN94]), but unification times never show up in profiles.

The improvement of discrimination trees is caused by the fact that one can use backtracking instead of trying the terms in the database one by one. With backtracking one can reuse initial segments of runs of the matching/unification algorithm. It is important to note that the improvement is not caused by the fact that parts of the data are combined, but by the reuse of initial segments of runs of the algorithm. In order to have advantage from the discrimination tree, the algorithm should process the term from left to right. When the terms cannot be processed by the algorithm in one left-right traversal, there is no point in using discrimination trees. If this is not possible one should either change the datastructure or change the algorithm. In [Gr94a] the datastructure is changed. In [dN94] the occurs check of the unification algorithm is modified in order to make it more appropriate for left-right traversal.

Storage of variables in a discrimination tree needs special attention. If one stores the different variables, the tree is called *perfect*. If one does not store the different variables, the tree is called *imperfect*. In the latter case the retrieval algorithm can be simpler, because it needs to compare only the structure of the terms, and it need not check that the same value is assigned to different occurrences of the same variable. However the algorithm will retrieve too many terms, and it is necessary to aftercheck the retrieved terms. In the case of an

imperfect discrimination tree, the retrieval algorithm will match $p(X, X)$ into $p(a, b)$.

In order to allow structure sharing between terms that are renamings of each other, one should normalize the terms before storing them. A term $t$ is *normal* if in its prefix representation variable $V_i$ occurs only after all variables $V_j$ with $j < i$. Terms that are not-normal can be decomposed into a normal term and a renaming substitution. For example $p(V_1, V_0, V_2)$ can be composed into $p(V_0, V_1, V_2)$ and $\{V_0 := V_1,\ V_1 := V_0,\ V_2 := V_2\}$. The substitutions can be stored at the leafs of the tree.

We have compared different ways of implementing discrimination trees in **disctree.c**. We always store terms in a normalized way. We tested only the retrieval algorithm for finding terms that can be matched into the given term, because insertion and deletion never show up in profiles. Before we discuss alternative datastructures for discrimination trees, we first give the retrieval algorithm. We use the fact that the variables in the tree are normalized, so we can use shallow binding without having an initialization problem. The algorithm given below is recursive. The initial call should be *findmatch*$(D, t, 1, f)$, in which $f$ is a function that is to be called when a solution is found. It will be called as $f(\Theta, R)$, where $\Theta$ is the matching constructed and $R$ is some type of reference to the term retrieved. The recursive calls have form *findmatch*$(D, t, i, f)$. Here $D$ is the subtree (actually a discrimination forest) from which the algorithm tries to retrieve the rest of term $t$ while it progressed already up to position $i$. (in the prefix representation of $t$.) There are 3 types of alternatives in $D$ : **(1)**. Variables $V_j$ that occurred already before. It should be checked that $V_j$ is bound consistently with **first**$(t, i)$. **(2)**. The variable $V_i$, occurring for the first time. $V_i$ should be assigned **first**$(t, i)$ to. **(3)** A functor/constant $f$. It should be checked that $f = t_i$. Procedure *findmatch* checks the alternatives 1,2,3 in increasing order. For Alternative 3, tail recursion can be used. This was done in all implementations. **first** and **rest** rely on the fact that the term $t$ is in prefix notation. **rest**$(\ t, i)$ can be easily obtained if $t$ is in prefixwe notation. First we give some notation, then the algorithm:

| | |
|---|---|
| $D$ | A discrimination tree. |
| **nil** | The empty discrimination tree. |
| $D[f]$ | The subtree of $D$ that belongs to operator/variable $f$. |
| $\Theta$ | A substitution. |
| $\#\Theta$ | The size of $\Theta$. |
| $\Theta[V]$ | The value of $\Theta$ for variable $V$. |
| $t$ | A term in prefix representation. |
| $\#t$ | The length of $t$ in prefix representation. |
| $t_i$ | The $i$-th operator/variable of $t$. |
| **first**$(t, i)$ | The subterm of $t$ that starts on position $i$. |
| **rest**$(t, i)$ | The index in $t$ of the subterm immediately after **first**$(\mathbf{t}, \mathbf{i})$. |

```
procedure findmatch(D, t, i, f)
if i = #t then
   For each solution in D call f(Θ, R).
else
begin
  for i := 1 to #Θ do
  begin
    if D[V_i] ≠ nil then
    begin
      if first(t, i) = Θ[V_i] then
          findmatch(D[V_i], t, rest(t, i), f)
    end
  end
  l := #Θ
  if D[V_l] ≠ nil then
  begin
    Θ[V_e] := first(t, i)
    #Θ := #Θ + 1
    findmatch(D[V_l], t, rest(t, i), f)
    #Θ := #Θ − 1
  end
  f := t_i
  if f is not a variable ∧ D[f] ≠ nil then
      findmatch(D[f], t, i + 1, f)
end
```

We now sum up the different ways of representing discrimination trees that we implemented:

**Deep Discrimination Trees**

In deep discrimination trees, the alternatives are stored in linked lists. The main disadvantage of this method is that, when there are many alternatives in one node, they all have to be tried one by one. It is quite well possible that this walking through the linked lists dominates the retrieval times.

**Shallow Discrimination Trees**

In shallow discrimination trees the linked lists are replaced by arrays of references to the subtrees. In this way the subtree belonging to a certain operator can be found at once, without having to search for it. Although shallow discrimination trees overcome the problems of deep discrimination trees, they introduce a couple of new problems: The first problem is the potentially large memory use. If there are many operators, and most nodes in the tree are sparse, then large amounts of memory are wasted for storing empty subtrees. Typically a tree has a small percentage of nodes with many alternatives near the root, and the vast majority of nodes has only 1 or 2 alternatives. Another problem is the fact that it is hard to extend the symbol table during the run of a theorem prover, because then the tree has to be reorganized. Some implementations of the splitting rule

introduce new symbols at run time. Finally, if there is a large set of predefined operators, (as is the case in Bliksem), these consume space in the array.

### Hashed Discrimination Trees

Hashed discrimination trees are designed as an attempt to overcome the last two problems of shallow discrimination trees. A node consists of an array of (references to) deep lists containing the subtrees. A hash function on the operator is used as index. The number of entries in the hash-array should be large enough to keep the linked lists short.

In the benchmarks two optimizations were used. First deep lists are always sorted. This saves some retrieval time. Second, we use shallow/hashed nodes only for nodes that have more than a certain number of alternatives, typically 10. When the number of a shallow/hashed node sinks below 5 it is changed back into a deep node. The difference avoids frequent reorganizations of nodes. We used the following test sets:

**S1** Set S1[n] consists of all terms that can be built over the following signature: There is one symbol $f$ with arity 3, and there are $n$ symbols $c_0, \ldots, c_{n-1}$ with arity 0. This makes the total number of terms equal to $n^3$. All terms in S1 are ground.

**S2** Set S2[n] is built up analogeously to S1, but with variables in addition to the constants. There is one symbol $f$ with arity 3. There are $n$ symbols $c_0, \ldots, c_{n-1}$ with arity 0, and 3 variables $V_0, V_1, V_2$. This makes the total number of terms equal to $(n+3)^3$. Some of the terms are renamings of each other, there are for example $f(V_0, V_1, V_2)$ and $f(V_2, V_1, V_3)$.

**S3** Set S3[n] is intended to have realistic characteristics. There is one symbol $f$ with arity 3. The elements of $S3[n]$ have form $f(X, Y, Z)$, where $X$ is from $c_0, \ldots, c_{n-1}$, $Y$ is from $c_0, c_1$, and $Z$ is from $c_0, c_1$. This makes the number of terms in $S3[n]$ equal to $4n$.

**Retrieval Times in seconds for S1**

| Set | nr. calls | time deep | time shallow | time hash [32] | naive |
|---|---|---|---|---|---|
| S1[10] | 1000 | 0.0015 | 0.0016 | 0.0014 | 0.0 |
| S1[20] | 8000 | 0.0132 | 0.0064 | 0.0086 | 12.0 |
| S1[30] | 27000 | 0.0566 | 0.0213 | 0.0289 | 148.0 |
| S1[40] | 64000 | 0.1662 | 0.0573 | 0.0817 | 774.0 |
| S1[50] | 125000 | 0.4856 | 0.1469 | 0.2002 | 3705.0 |
| S1[60] | 216000 | 1.2977 | 0.2669 | 0.3554 | 18907 |
| S1[70] | 343000 | 2.8589 | 0.4245 | 0.5741 | 51485 |
| S1[80] | 512000 | 5.4643 | 0.6324 | 0.9215 | 114805 |
| S1[90] | 729000 | 10.2555 | 0.9055 | 1.2594 | unknown |

**Retrieval Times in seconds for S2**

| Set | nr. calls | time deep | time shallow | time hash [32] |
|---|---|---|---|---|
| S2[10] | 2197 | 0.02 | 0.02 | 0.02 |
| S2[20] | 12167 | 0.19 | 0.16 | 0.17 |
| S2[30] | 35937 | 0.61 | 0.51 | 0.52 |
| S2[40] | 79507 | 1.56 | 1.16 | 1.23 |
| S2[50] | 148877 | 3.51 | 2.23 | 2.35 |
| S2[60] | 250047 | 6.87 | 3.72 | 4.01 |
| S2[70] | 389017 | 12.13 | 5.77 | 6.28 |
| S2[80] | 571787 | 19.97 | 8.49 | 9.32 |
| S2[90] | 804357 | 34.06 | 11.94 | 13.34 |

**Retrieval Times in $10^{-3}$ seconds for S3**

| Set | nr. calls | time deep | time shallow | time hash [32] |
|---|---|---|---|---|
| S3[10] | 40 | 0.04 | 0.03 | 0.03 |
| S3[20] | 80 | 0.09 | 0.06 | 0.07 |
| S3[30] | 120 | 0.15 | 0.10 | 0.11 |
| S3[40] | 160 | 0.22 | 0.14 | 0.16 |
| S3[50] | 200 | 0.30 | 0.17 | 0.20 |
| S3[60] | 240 | 0.39 | 0.21 | 0.24 |
| S3[70] | 280 | 0.47 | 0.24 | 0.28 |
| S3[80] | 320 | 0.58 | 0.28 | 0.32 |
| S3[90] | 360 | 0.70 | 0.31 | 0.36 |

One can conclude from this that complicated optimizations of the datastructure for discrimination trees are mainly worth the effort if there are no, or very little variables. They made the implementation much harder.

If one wants more improvement it is probably necessary to isolate subtrees with special properties, e.g. ground, non-branching, as is done in [Wm95]. These subtrees then can be processed by specialized, optimized algorithms.

## 5 Subsumption

In this section we describe the subsumption algorithm in Bliksem. Our main improvement is in the clause-to-clause algorithm. (Terminology from [Tam98]) A clause $c$ subsumes a clause $d$ if there is a substitution $\Theta$, such that $c\Theta \subseteq d$. If this is the case then $d$ is redundant and can be deleted. Subsumption is usually distinguished into two types: If one checks for a newly derived clause whether or not it is subsumed by an existing clause, this is called the *forward* check. If one checks whether or not a newly derived clause subsumes an existing clause, this is called the *backward* subsumption check. For the forward subsumption check we use discrimination trees, and bit filters in order to restrict the set of candidates that have to be checked. We have no special optimizations for backward subsumption. It is generally agreed upon that backward subsumption is not as important as forward subsumption, and not as time critical. [Tam98], [Wm95].

The forward subumption check is very time consuming, if not carefully implemented, because it has to be applied on every generated clause. The backward subsumption check is done only on the clauses that are kept. There are usually much more generated clauses, than kept clauses.

When optimizing the forward subsumption check, we distinguish two types of optimizations: *Local* optimizations are optimizations on the subsumption check for two individual clauses $c$ and $d$. *Global* optimizations are optimizations exploiting the fact that one has many clauses on the place of $c$. (or $d$)
The problem of deciding whether or not a clause $c$ subsumes a clause $d$ is $NP$-complete. However, the hard cases are rare, so one should not make time-costly optimizations. In particular in [GL85] it is proposed to split a clause into connected components before doing the subsumption check. This certainly improves the complexity when the clause is complicated. Nevertheless most clauses are non-complicated, so one needs a way to exploit the existence of components, without having to compute them. Our algorithm does exactly this. It uses straightforward backtracking, but on a failure it marks the literals that contributed to the failure. Since a literal can never cause a failure in another component there is never backtracking between different components.

**Definition 1.** *Let $A$ and $B$ be literals. $A$* matching *of $A$ into $B$ is a substitution $\Theta$ for which $A\Theta = B$. When we write* the matching *of $A$ into $B$ we mean the $\subseteq$-minimal matching, which is unique. Two substitutions $\Theta_1$ and $\Theta_2$ are* consistent *if there is no variable $V$, for which $V\Theta_1 \neq V\Theta_2$.*

When we are discussing whether or not $c$ subs $d$, we treat the variables in $d$ as constants, so when we write 'variable' we mean a variable of $c$.

**Theorem 1.** *Let $c_1 = \{A_1, \ldots, A_p\}$ and $c_2 = \{B_1, \ldots, B_q\}$ be clauses. The following two statements are equivalent:*

1. *There is a substitution $\Theta$, such that $c_1\Theta \subseteq c_2$.*
2. *For each $A_i$ there is a matching $\Theta_i$ such that*
   (a) *$\Theta_i$ is the matching of $A_i$ into a literal $B_{\lambda_i}$ with $1 \leq \lambda_i \leq q$,*
   (b) *each pair of matchings $\Theta_{i_1}$ and $\Theta_{i_2}$ is consistent, $1 \leq i_1, i_2 \leq p$.*

**Definition 2.** *Let $c$ be a clause. A* component partition *consists of clauses $c_1, \ldots, c_n$, s.t. $c = c_1 \cup \cdots \cup c_n$, and whenever $i \neq j$, then there are no overlapping variables between $c_i$ and $c_j$. (This implies that the $c_i$ are disjoint) The component partition is* maximal *if $n$ is maximal.*

Components are important because of the following, see [GL85].

**Lemma 1.** *Let $c_1, \ldots, c_n$ be a component partition.*
*$c_1 \cup \cdots \cup c_n$ subs $d$ iff $c_1$ subs $d, \ldots, c_n$ subs $d$.*

This follows from Theorem 1.

The algorithm in [GL85] is based on this. When checking if $c$ subs $d$ one first tries to split $c$ into components. If this is possible, the components are checked

independently. If it is not possible to split, then one tries to match one literal $A \in c$ into $d$. If this is possible one recursively checks for $(c\backslash\{A\})\Theta$ subs $d$. Because $(c\backslash\{A\})\Theta$ has been instantiated there may be more components in $(c\backslash\{A\})\Theta$. In [GL85] it is argued that one usually does the subsumption check $c$ subs $d$ with many clauses in the place of $d$, and only one clause in the place of $c$, and that therefore it is harmless to compute the components of $c$. This is the case only for the backward subsumption check. Before we can explain our own algorithm we need a technical notion:

**Definition 3.** *Let $A$ be a literal, and let $c_2 = \{B_1, \ldots, B_q\}$ be a clause. The* matching disjunction *of $A$ into $\{B_1, \ldots, B_q\}$ is the disjunction $\Theta_1 \vee \cdots \vee \Theta_q$, with $\Theta_i$ defined as the matching of $A$ into $B_j$ if it exists, and $\Theta_i = \bot$ otherwise.*

*Let $c_1 = \{A_1, \ldots, A_p\}$ and $c_2 = \{B_1, \ldots, B_q\}$ be clauses. The* matching disjunction set *is the set of matching disjunctions for the $A_i$.*

Our algorithm is based on the fact that, using the matching disjunctions, the subsumption check is essentially the same as testing for unsatisfiability in propositional logic. One could define a resolution rule on matching clauses, and use ordered resolution to solve the subsumption problem, but it is better to use semantic tableaux. The first step is to write the matching clauses as sets, and to delete the $\bot$ from the clauses.

**Definition 4.** *A* matching clause *is a set of substitutions $\{\Theta_1, \ldots, \Theta_q\}$. A* model *of a matching clause $c$ is a set of pairwise consistent substitutions $M$, such that $c \cap M \neq \emptyset$. A model of a matching clause set $C$ is a model of the clauses $c \in C$.*

**Theorem 2.** *Let $c$ and $d$ be clauses. Let $C$ be the matching clause set of $c$ into $d$. We have $c$* subs *$d$ iff $C$ has a model.*

*Example 1.* Consider the clauses $c = \{p(X), q(X, Y), r(Y)\}$ and $d = \{p(a), p(b), q(b, a), r(a), r(c)\}$. The set of matching clauses equals:

$$\{\{X := a\}, \{X := b\}\}, \{\{X := b, Y := a\}\}, \{\{Y := a\}, \{Y := c\}\}.$$

It is the case that $c$ subsumes $d$ and there is a model of the clause set:

$$M = \{\{X := b\}, \{X := b, Y := a\}, \{Y := a\}\}.$$

The following algorithm checks whether or not a set of matching clauses has a model. For the matching that it chooses at the $i$-th level, it remembers whether or not this matching played a role in later conflicts in the variable $\phi_i$. The algorithm backtracks only on matchings that contributed to a later conflict. In this way the effect of splitting into components is obtained, since there can be no conflict between matchings that belong to different components.

```
procedure backtrack(i)
begin
  (Heuristically) choose a clause and call it d_i.
  for Θ_i ∈ d_i do
  begin
    if there is a Θ_j with 1 ≤ j < i, for which Θ_i and Θ_j
     are inconsistent then
         φ_j := t for the lowest such j
    else
    begin
      φ_i := f
      backtrack(i + 1)
      if φ_i = f then return
    end
  end
end
```

## References


[Bl00] The benchmarks and Bliksem scan be obtained from the page from which this paper was obtained.

[Bb74] E.C. Berkely, D.G. Bobrow (editors), The Programming Language LISP: Its Operation and Applications, The MIT Press, Cambridge, Mass. 1974.

[Wm95] A. Buch, Th. Hillenbrand, Waldmeister: Development of a High Performance Completion-Based Theorem Prover, SEKI-Report SR-96-01, 1995.

[Ch93] J. Christian, Flatterms, Discrimination Nets, and Fast Term Rewriting, Journal of Automated Reasoning 10, pp. 95-113, Kluwer Academic Publishers, 1993.

[dN94] H. de Nivelle, An Algorithm for the Retrieval of Unifiers from Discrimination Trees, Journal of Automated Reasoning 20, pp. 5-25, Kluwer Academic Publishers, 1998.

[Gr94a] P. Graf, Substitution Tree Indexing, Internal Report MPI-I-94-251, Max Planck Institut für Informatik, Saarbrücken, 1994.

[Gr94b] P. Graf, Extended Path-Indexing, in CADE 12, Ed. Alan Bundy, pp. 514-528, 1994.

[GL85] G. Gottlob, A. Leitsch, On the Efficiency of Subsumption Algorithms, Journal of the Association for Computing Machinery, Vol. 32, No. 2, pp. 280-295, 1985.

[McC92] W. McCune, Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval, Journal of Automated Reasoning, 9(2):147-167, 1992.

[McC99] W. McCune, the Otter Theorem Prover, can be obtained from `http://www-unix.mcs.anl.gov/AR/otter`

[Spass99] C. Weidenbach, the Spass Theorem Prover, can be obtained from `http://spass.mpi-sb.mpg.de`, 1999

[Tam98] T. Tammet, Towards Efficient Subsumption, in CADE 15, LNAI 1421, Springer Verlag, pp. 427-441, 1998.

[Vor95] A. Voronkov, The Anatomy of Vampyre, Implementing Bottom-Up Procedures with Code Trees, Journal of Automated Reasoning 15, pp. 237-265, 1995.