An Implementation Technique for a Class of Bottom-Up Procedures

Andrei Voronkov[1]*

Computing Science Department
Uppsala University
voronkov@csd.uu.se

Abstract. We present an implementation technique for a class of bottom-up logic procedures. The technique is called *code trees*. It is intended to speed up most important and costly operations, such as subsumption and resolution. As an example, we consider the forward subsumption problem which is the bottleneck of many systems implementing first order logic.
In order to efficiently implement subsumption, we specialize subsumption algorithms at run time, using the *abstract subsumption machine*. The abstract subsumption machine executes subsumption using sequences of instructions that are similar to the WAM instructions [War83]. It gives an efficient implementation of the "clause at a time" subsumption problem. To implement subsumption on the "set at a time" basis we combine sequences of instructions in *code trees*.
We show that this technique yields a new way of indexing clauses. Some experimental results are given.
The code trees technique may be used in various procedures, including OLDT-resolution, SLD-AL-resolution, bottom-up evaluation of logic programs and disjunctive logic programs.

1 Introduction

There is a growing interest in bottom-up algorithms for executing logic programs. It is mainly connected with the development of the deductive databases technology. For instance, various modifications of magic sets transformations require the semi-naive bottom-up evaluation of the transformed logic program [BMSU86, BR87]. SLD-AL resolution [Vie89] and OLDT-resolution [TS86, War92] are further examples of systems which combine the bottom-up and top-down evaluations.

Bottom-up algorithms have been used through the years in automated deduction. Bottom-up algorithms are used by many automatic theorem proving procedures, e.g., binary resolution [Rob65], hyperresolution [Rob65a] and the inverse method [Mas67, Vor92]. In order to efficiently implement such procedures, indexing techniques have been develop [Sti89, McC88, McC92]. Indexing allows main operations (like resolution and subsumption) to be implemented on the "set at a time" basis. As noted in [Wos92], the implementation of discrimination trees in the well known theorem prover OTTER resulted in a much faster performance.

In this paper we consider an alternative implementation technique which may be used for a large variety of bottom-up algorithms. These algorithms are based on the following loop:

* Supported by Swedish TFR grant no. 93-409

Let D be the initial database of clauses.

```
loop:
```

1. Apply given inference rules to D (or a part of D). Let ΔD be the new clauses.
2. If $D, \Delta D$ satisfy a termination criterion, then terminate.
3. Apply a retention test to ΔD.
4. Apply a retention test to D, based on ΔD.
5. Add ΔD to D.

```
end of loop.
```

For this class of procedures, we call clauses in D *kept* clauses, and clauses in ΔD *new* clauses.[2] Let us consider these steps in more details.

1. Inference rules are usually resolution rules applied to clauses from D. In SLD-AL-resolution inference rules and clauses are more complicated. In fact, it supports two kinds of clauses, the first represents queries and the second represents solutions to the queries.
2. There are many kinds of the termination criterion. Usually, the procedure terminates when a goal is found. The termination criterion can also be based on the *saturation*: the procedure terminates when no new clauses can be generated.
3. The retention test for ΔD usually includes *forward subsumption*: remove from ΔD every clause subsumed by a clause in D. In principle, there might be other criteria for discarding new goals [McC90]. When the number of generated clauses is big compared to the number of kept clauses, forward subsumption becomes the most costly operation.
4. The retention test for D is usually based on *back subsumption*: remove from D every clause subsumed by a kept clause in ΔD.

This loop is used in most implementations of binary resolution and hyperresolution [Rob65, Rob65a], the inverse method [Mas67, Vor92], implementations based on magic sets [BMSU86, BNRST87], tabled computations of logic programs [TS86, War92, SR93], various deductive databases procedures, for example query answering and integrity checking [LT85, LST86, Dec86, Das92]. Top-down procedures which use lemmaizing, like SLD-AL-resolution [Vie89] or some implementations of model elimination [AS92] also use a similar scheme. Such kind of procedures is often needed in implementations of the static analysis or the abstract interpretation of logic programs [BGL93, CD93]. In this case clauses are Horn clauses, not necessarily facts.

In [WOL91] these procedures are characterized as procedures which *retain information* during query answering. It is necessary to retain clauses and to use subsumption, in order to avoid a combinatorial explosion of the database D. Prolog, on the contrary, considers only the current branch of the search tree and thus retains no information about the previously exploited branches.

The main problems faced by bottom-up procedures are based on the necessity to efficiently handle large dynamically changing databases of clauses. Implementations of Prolog face very different problems. Goals in Prolog are executed on the "goal at a time" basis, which requires very fast processing of the current goal. WAM which has become a standard for implementing Prolog, is hence designed with the aim of the efficient execution of one (current) subgoal or a chain of subgoals at a time. Such "goal at a time" processing is inadequate for bottom-up algorithms which must often handle millions of clauses. The following features are crucial in understanding the

[2] Our terminology *kept* and *new* is taken over from the one used in the OTTER theorem prover [McC90]

difference between implementations of bottom-up procedures and implementations of Prolog.

1. Subsumption should often be called (considerably) more times than resolution. Thus, one has to pay attention to the implementation of subsumption.
2. The implementation of resolution in the WAM style is not efficient any more due to the following reasons:
   (a) The set of clauses is changing dynamically. This makes it impossible to precompile clauses which are to be used as inference rules.
   (b) Inference rules may involve *several* dynamic database clauses.
   (c) The database D can be very large, with non-ground clauses. It makes the usual indexing methods inadequate.

Thus, in order to efficiently implement bottom-up procedures, one needs to efficiently implement resolution (or similar inference rules) and subsumption (both forward and back) for *large dynamically changing* sets of clauses.

In this paper we introduce an implementation technique called *code trees* which is intended for speeding up these basic procedures. We shall show its use it on the forward subsumption problem. The idea of code trees is very general and may as well be applied to various concepts of resolution. It can also be used for the implementation of equality reasoning procedures, like paramodulation [RW69] and rewriting.

Our technique is based on the following two main ideas. The first idea is to compile specialized subsumption procedures for kept clauses. It can be considered as a run time specialization of a general subsumption algorithm. This technique has much in common with the technique of WAM-based Prolog implementations. We call it *the abstract subsumption machine.* It gives a very efficient subsumption algorithm for the problem of subsuming a clause by a clause. A specific feature of the use of this approach is that the compilation must be performed at run time.

When the set D of kept clauses is large, the implementation of subsumption on the "clause at a time" basis becomes inefficient. In our theorem proving experiments we ran an example which required about $25 * 10^9$ subsumption checks on deep terms. Such number of subsumption checks performed on the "clause at a time" basis is not feasible for any modern computer. In order to perform subsumption on the "set at a time" basis we introduce code trees which merge several specialized algorithms into one.

2 Subsumption. The abstract subsumption machine

We assume acquaintance with the basic notions of terms and substitutions. We use the notion of a clause adopted in the theorem proving community: a *clause* is a list of terms. Clauses used in logic programming can be interpreted in our terminology in the standard way:

1. *Fact* A is interpreted as A.
2. *Disjunctive fact* $A_1 \vee \ldots \vee A_n$ is interpreted as $A_1, \ldots, A_n$.
3. *Rules* $A\ :-\ A_1, \ldots, A_n$ are interpreted as $A, \neg A_1, \ldots, \neg A_n$.
4. *Disjunctive rules* $B_1 \vee \ldots \vee B_m\ :-\ A_1, \ldots, A_n$ are interpreted as $B_1, \ldots, B_m, \neg A_1, \ldots, \neg A_n$.
5. *Goals* $:-\ A_1, \ldots, A_n$ are interpreted as $\neg A_1, \ldots, \neg A_n$.

Thus, we consider negation $\neg$ as an ordinary function symbol. Terms in a clause are also called *literals*. *Variables* will be denoted by the lower case letters x, y, z, u, v, w, maybe with indices. Let c_1, c_2 be two clauses. Clause c_1 *subsumes* clause c_2 iff there is a substitution θ such that $c_1\theta$ is a subset of c_2[3]. For example, let $c_1 = P(x, f(y)), P(z, f(f(z)))$ and $c_2 = P(y, f(f(y)))$. Then c_1 subsumes c_2 with the substitution $[x/y, y/f(y), z/y]$ and c_2 subsumes c_1 with the substitution $[y/z]$.

A clause is a *unit clause* iff it consists of exactly one literal. Clauses with more than one literal will be called *multi-literal* clauses. In most logic programming applications, the bottom-up procedures use only unit clauses. Multi-literal Horn clauses are used in the bottom-up abstract interpretation of logic programs. Disjunctive clauses are used in disjunctive logic programs and in theorem proving procedures. **In this section, we only consider unit clauses.** Multi-literal clauses will be considered in Section 4.

The algorithms considered in this paper deal with *positions* in clauses. A position in a clause c corresponds to an occurrence of a subterm in c. A tree representation of the (unit) clause $P(g(x_0, x_1), f(a, x_0, x_2))$ and the corresponding set of positions are given in Figure 1. This clause will be our running example until the end of Section 3.

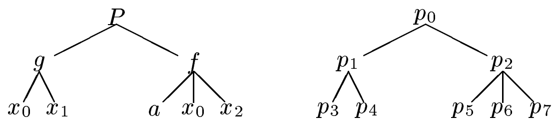

Fig. 1. The clause $P(g(x_0, x_1), f(a, x_0, x_2))$ and its set of positions $\{p_0, \ldots, p_7\}$.

In order to present our algorithms, we need to introduce a few operations on positions. Let p be a position, corresponding to a list of arguments $t_1, \ldots, t_n$. $right(p)$ is the position to the right of p, i.e. the one that corresponds to $t_2, \ldots, t_n$. Otherwise, $right(p)$ is undefined. For example, on Figure 1 we have $right(p_1) = p_2$, $right(p_5) = p_6$ and $right(p_0)$ is undefined.

If the position p corresponds to a term t then $down(p)$ denotes the position of the first argument of t. If t has no arguments (i.e. t is a variable or a constant) then $down(p)$ is undefined. For example, on Figure 1 we have $down(p_0) = p_1$, $down(p_1) = p_3$ and $down(p_6)$ is undefined.

$var(p)$ is true iff the term at the position p is a variable. $functor(p)$ denotes the principal functor of the term in the position p. For simplicity, we assume that the arity of each function symbol is fixed. For example, terms $g(x)$ and $g(x, y)$ cannot be used together[4]. $defined(p)$ is true iff p is a valid position in the clause (operations $down, right$ may lead to invalid positions). For p, q positions, $p == q$ is true iff terms at positions p, q are equal. $\neq\neq$ is the negation of $==$. For example, on Figure 1 $var(p_0)$ is false, $var(p_7)$ is true, $functor(p_0) = P$, $p_3 == p_6$ is true and $p_3 == p_1$ is false.

[3] I.e. every literal in $c_1\theta$ is a literal in c_2

[4] In logic programming languages such use of functors is allowed. In this case we have to consider pairs g/n consisting of a function symbol and its arity. We do not treat this case for the sake of simplicity.

For unit clauses c_1, c_2, the subsumption problem reduces to the *matching problem*: find a substitution θ such that $c_1\theta = c_2$. An example of the matching algorithm *match* is given in Figure 2. We do not claim this algorithm to be the most efficient. One can often do better, depending on the nature of the problem and the concrete representation of clauses. The algorithm *match* is given in an abstract form. It may be adapted to all concrete representations of terms I am aware of, for example in the one used in logic programming and those used in automated deduction.[5]

```
Algorithm match(c1, c2)
  Let p, q be positions
  Let subst, post be initially empty
  stacks of pairs of positions
begin
  push(⟨c1, c2⟩, post);
  while nonempty(post)
  {
    ⟨p, q⟩ := pop(post);
    if var(p)
    then push(⟨p, q⟩, subst)
    else if var(q) or functor(p) ≠ functor(q)
    then exit with fail
    else
    {
      if right(p) defined
      then push(⟨right(p), right(q)⟩, post);
      if down(p) defined
      then push(⟨down(p), down(q)⟩, post)
    }
  }
  forall cliques c = ⟨p0, q0⟩, ..., ⟨pn, qn⟩ ∈ subst
  forall i ∈ {1, ..., n} do
  {
    if q0 == qi
    then remove ⟨pi, qi⟩ from subst
    else exit with fail
  }
  exit with success
end
```

In this algorithm the stack *subst* represents the matching substitution. The stack *post* is used to keep the pairs of positions whose matching is postponed. p, q are used to denote the positions which are currently under consideration.
For a stack S and an element e, $push(e, S)$ denotes the operation of adding e at the beginning of S. To add e to the end of the stack S, we write $add(e, S)$. $pop(S)$ removes the first element from the stack S and returns this element as the result. $nonempty(S)$ is true if the stack S is not empty.
Let $S = \langle p_0, q_0\rangle, \ldots, \langle p_n, q_n\rangle$, where p_i are positions. Then $tail(S)$ is $\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle$ A *clique* in S is any maximal subsequence $\langle p_{i_0}, q_{i_1}\rangle, \ldots, \langle p_{i_k}, q_{i_k}\rangle$ of S such that $k \geq 1$ and $p_{i_j} == p_{i_0}$ for all $j \in \{0, \ldots, k\}$.

Fig. 2. The general matching algorithm for clauses c_1, c_2

The algorithm *match* tries to construct a substitution θ such that $c_1\theta = c_2$. First of all, *match* traverses the two terms, putting into *subst* pairs $\langle v, t\rangle$, where v is a variable in c_1, t a term in c_2. Then it compares terms in c_2 corresponding to different

[5] By representations used in automated deduction I mean two representations of terms with pointers, in one of them the references to terms and references to term lists are the same (like in OTTER), in the other a list of terms contains a reference to the first term in the list (Vampire). Both can be used with structure sharing [BM72] or without. In automated deduction a list of terms usually contains a reference to the tail of the list. In logic programming, the list of terms is usually an array of references to terms.

occurrences of the same variable in c_1. There were two reasons for us to delay the comparison. The first reason is that the term comparison is potentially very costly operation. The second reason is that this form of the algorithm makes more *code sharing* considered below in Section 3.

Theorem 1. *The algorithm match is sound and complete. More precisely:*

1. *If c_1 does not subsume c_2 then $match(c_1, c_2)$ terminates with failure.*
2. *If c_1 subsumes c_2 then $match(c_1, c_2)$ terminates with success. In this case after termination subst contains the matching substitution.*

All proofs in this paper are omitted due to the lack of space. They may be found in [Vor94].

The set of clauses in bottom up procedures is dynamically changing. Thus, it is not possible to pre-compile the subsumption algorithm. However, one can try to compile subsumption algorithms for kept clauses c at run time, in order to execute subsumption more efficiently. The compilation requires some time, but it can improve the performance in general, if the subsumption algorithm between c and other clauses will be called several times. Since clauses c_1, c_2 in $match(c_1, c_2)$ play different roles, there are two ways to specialize the subsumption algorithm *match*. If c_1 is given and c_2 is unknown, we obtain a specialized algorithm $smatch_{c_1}(c_2)$. It used for the subsumption of new clauses by c_1 (forward subsumption). In the other case, if c_2 is given and c_1 is unknown, we obtain a specialized algorithm $smatch_{c_2}(c_1)$. It is used to subsume c_1 by new clauses (back subsumption). The two specializations are quite different. For the simplicity and due to the lack of space, we shall only consider forward subsumption.

We shall represent the specialized matching algorithms as a sequence of instructions of an abstract machine. All possible instructions form "the building blocks" of matching algorithms. For the reasons that will be clear in Section 3 it is preferable that the instruction set be as simple as possible. We shall show an example how to produce these instruction sets for the matching problem.

Let $c_1 = P(g(x_0, x_1), f(a, x_0, x_2))$. When we partially evaluate *match* with respect to c_1 as the first argument, we obtain the specialized algorithm shown on Figure 3.[6]

Each line of this specialized algorithm is an instruction of our *abstract subsumption machine*. There are 9 different kinds of instructions (Initialize, Check, Down, Right, Push, Pop, Put, Compare, Success). They are displayed in the right column of Figure 3. Some of instructions have parameters (Check, Push, Pop, Put and Compare).

One can imagine the abstract subsumption machine as a kind of Turing machine. q plays the role of the head of the machine. Instead of moving along the tape, the head q moves on the set of positions in the clause. It uses two arrays: *post* to record postponed positions and *subst* for keeping tracks of terms substituted for variables. The meaning of the set of instructions is the following:

[6] We also perform an obvious optimization. Instead of the sequence of instructions

$push(E, post)$
$q := pop(post)$

where E is an expression, we use the equivalent $q := E$.

Algorithm $match_{P(g(x_0,x_1),f(a,x_0,x_2))}(c)$	
Let q be a position	
Let $subst, post$ be initially empty arrays of positions	
begin	
$q := c$;	Initialize
if $var(q)$ **or** $functor(q) \neq P$ **then** exit with fail;	Check P
$q := down(q)$;	Down
if $var(q)$ **or** $functor(q) \neq g$ **then** exit with fail;	Check q
$post[0] := right(q)$;	Push 0
$q := down(q)$;	Down
$subst[0] := q$;	Put 0
$q := right(q)$;	*Right*
$subst[1] := q$;	*Put 1*
$q := post[0]$;	Pop 0
if $var(q)$ **or** $functor(q) \neq f$ **then** exit with fail;	Check f
$q := down(q)$;	Down
if $var(q)$ **or** $functor(q) \neq a$ **then** exit with fail;	Check a
$q := right(q)$;	Right
$subst[2] := q$;	Put 2
$q := right(q)$;	*Right*
$subst[3] := q$;	*Put 3*
if $subst[0] \neq\neq subst[2]$ exit with fail;	Compare 0 2
exit with success	Success
end	

Fig. 3. The specialized matching algorithm for the clause $P(g(x_0, x_1), f(a, x_0, x_2))$.

In the right column there are instructions of the abstract subsumption machine. Unnecessary instructions are shown in italic, like *Right*.

Initialize	Set q to the initial position (the term c)
Check P	Check that $functor(q)$ is P
Down	Go down the current position (i.e. from a term to its arguments)
Right	Go to the right (i.e. from an argument in a term to its next argument)
Push n	Push the position to the right on the nth position in *post*
Pop n	Set q to the nth position in *post*
Put n	Put the current position on the nth position in *subst* (as the substitution for the nth variable).
Compare m n	Compare terms at positions m, n in *subst* (both positions correspond to the substitution for the same variable).
Success	Exit with success.

If a *Compare* or a *Check* instruction fails then the whole algorithm fails. We shall prove in Theorem 2 below that these 9 types of instructions are enough to specialize $match(c_1, c_2)$ for any c_1. The proof will use a specializing algorithm.

As one can see, the instructions on Figure 3 displayed in italic are unnecessary: they put in *subst* values which will never be used. This suggests the following definition. A position p in the clause c is *unimportant* iff it is the position of the variable u_1 in a subterm $f(t_1, \ldots, t_m, u_1, \ldots, u_n)$ of c and $u_1, \ldots, u_n$ have unique occurrences in c. All other positions in c are *important*.

The specializing algorithm *matchspec* accepts a term c_1 as the input and constructs the instruction sequence for $smatch_{c_1}(c_2)$. The algorithm *matchspec* is shown in Figure 4. The following theorem shows the soundness and correctness of the of the *matchspec* algorithm:

In the algorithm below $important(p)$ means that the position p is defined and important.

```
Algorithm matchspec(c)
  Let p be a position
  Let subst, post be initially empty stacks
    of pairs of the form ⟨position,number⟩
  Let instr be an initially empty
    instruction sequence.
  Let var_count, post_count := 0
begin
  add(Initialize, instr);
  p := c;
  loop
    if var(p)
    then
    {
      Add Put var_count to instr;
      Add ⟨p, var_count⟩ to subst ;
      var_count := var_count + 1
    }
    else Add Check functor(p) to instr
    if important(down(p))
    then
    {
      if important(right(p))
      then
      {
        Add Push post_count to instr;
        push(⟨right(p), post_count⟩, post);
        post_count := post_count + 1
      }
      Add Down to instr;
      p := down(p)
    }
    else if important(right(p))
    then
    {
      Add Right to instr;
      p := right(p)
    }
    else if nonempty(post)
    then
    {
      ⟨p, k⟩ = pop(post);
      Add Pop k to instr
    }
    else exit loop
  end of loop
  forall cliques
    c = ⟨p_0, k_0⟩, ..., ⟨p_n, k_n⟩ ∈ subst
  forall i ∈ {1, ..., n} add
    Compare k_0 k_i to instr
end
```

Fig. 4. The specializing algorithm $matchspec(c)$.

Theorem 2. *For all unit clauses* c_1, c_2 $matchspec(c_1)(c_2)$ *is equivalent to* $match(c_1, c_2)$.

In our early experiments we used the abstract subsumption machine in two ways. The first way is to create, for each kept clause c, a structure representing the sequence of instructions of $matchspec(c)$. Then, instead of performing subsumption on two terms c_1, c_2, we used an *interpreter* of the abstract machine. The interpretation could give from no speedup (ground terms) to the speedup of the order of 3-4 (clauses with about 10 occurrences of variables). The second way was to *compile* the abstract machine into the native code of our computer. On clauses with 10 occurrences of variables the speedup of the compilation approach was of the order of 10. This latter is a considerable improvement of subsumption performed on the "clause at a time" basis. However, the main idea behind the use of the abstract machine is much more powerful than that. The idea is to combine instruction sequences in *code trees.*

3 Code trees

It has been widely recognized in the deductive database research that the logic programming "goal at a time"[7] mode of evaluation is not adequate for handling large amounts of clauses. It is true both for rules producing new clauses and for subsumption algorithms. In this section we demonstrate how to perform subsumption on the "set at a time" basis using *code trees.*

The idea of code trees is simple but extremely powerful. In the previous section, we created a sequence of the abstract subsumption machine instructions in order to represent a specialized subsumption algorithm $matchspec(c)$ induced by a given clause c. Code trees create similar instructions in order to represent a specialized subsumption algorithm $Match_D$ induced by a given *set* of clauses D. Code trees can be considered as the abstract subsumption machine extended by several new instructions. We gave it a different name to stress that the instructions of $Match_D$ do not form a linear structure as the instructions of $matchspec(c)$. Rather, they are structured as *trees*.[8] Thus, we *dynamically create an algorithm for the forward subsumption problem induced by a dynamically changing database of clauses.*

Assume that we have a (large) database D of kept clauses and a new clause d. We have to check whether d is subsumed by a clause in D. There are at least two ways to perform subsumption by D on the "set at a time" basis. The first idea is to exploit the common structure of clauses in D in order to select a (hopefully small) subset of potential candidates for subsumption. This idea has given rise to the use of various indexing schemes, like discrimination trees [McC92] or path indexing [Sti89]. Indexing plays the role of a *filter* for the selection of potentially useful clauses. The second idea is to combine *algorithms* which perform subsumption by D in one algorithm (the code tree). In Section 5 we briefly compare the two approaches.

Let us start with an example. Consider two instruction sets, which represent forward subsumption algorithms for clauses $P(x, a, x)$ and $P(x, x, b)$. They are shown on Figure 5 on the left.

The first 5 instructions of the two sequences coincide. It means that two algorithms do the same job in the beginning. In order to not repeat the same job we can perform subsumption as follows. First of all, we execute the common 5 instructions. If one of these common instruction fails, then the algorithm fails. Then we execute the rest of the first instruction sequence. If it fails, we backtrack to the point where the two sequences diverge and execute the rest of the second instruction sequence. For such an algorithm to be sound we have to ensure that all state changes made after the backtrack point be restored after backtracking. To this end we add two additional instructions: $Fork$ and $Restore$. $Fork$ memorizes values of variables which

[7] in deductive databases it is also called "tuple at a time" since deductive databases consider relations as sets of tuples.

[8] In the case of multi-literal clauses, the specialized code is even more complicated than trees. For example, there are instructions which cause a potentially unlimited backtracking.

$\boxed{\begin{array}{c} \mathit{Initialize} \\ \mathit{Check}\ P \\ \mathit{Down} \\ \mathit{Put}\ 0 \\ \mathit{Right} \end{array}}$	$\boxed{\begin{array}{c} \mathit{Initialize} \\ \mathit{Check}\ P \\ \mathit{Down} \\ \mathit{Put}\ 0 \\ \mathit{Right} \end{array}}$	$\longrightarrow$	$\boxed{\begin{array}{c} i_0 :\ \mathit{Initialize} \\ i_1 :\ \mathit{Check}\ P\ i_2 \\ i_2 :\ \mathit{Down}\ i_3 \\ i_3 :\ \mathit{Put}\ 0\ i_4 \\ i_4 :\ \mathit{Right}\ i_5 \end{array}}$
$\begin{array}{c} \mathit{Check}\ a \\ \mathit{Right} \\ \mathit{Put}\ 1 \\ \mathit{Compare}\ 0\ 1 \\ \mathit{Success} \end{array}$	$\begin{array}{c} \mathit{Put}\ 1 \\ \mathit{Right} \\ \mathit{Check}\ b \\ \mathit{Compare}\ 0\ 1 \\ \mathit{Success} \end{array}$	$\begin{array}{c} i_5 :\ \mathit{Fork}\ j_5\ i_6 \\ i_6 :\ \mathit{Check}\ a\ i_7 \\ i_7 :\ \mathit{Right}\ i_8 \\ i_8 :\ \mathit{Put}\ 1\ i_9 \\ i_9 :\ \mathit{Compare}\ 0\ 1\ i_{10} \\ i_{10} :\ \mathit{Success} \end{array}$	$\begin{array}{c} j_5 :\ \mathit{Restore}\ j_6 \\ j_6 :\ \mathit{Put}\ j\ i_7 \\ j_7 :\ \mathit{Right}\ j_8 \\ j_8 :\ \mathit{Check}\ b\ j_9 \\ j_9 :\ \mathit{Compare}\ 0\ 1\ j_{10} \\ j_{10} :\ \mathit{Success} \end{array}$

Fig. 5. Two instruction sequences and the corresponding code tree.

Shared parts of code are put in a dashed box.

may have changed after having executed the rest of the first instruction sequence. *Restore* restores the values. We call such a pair *Fork-Restore* a *fork*.[9] The new set of instructions representing subsumption for this set of two clauses is shown on Figure 5 on the right. The instructions form a tree. Since the set of clauses D can be changing dynamically, each instruction, except for *Success*, must have an additional argument representing the next instruction. This argument may change when we add new clauses to the database D or remove clauses from D. Such sets of instructions represent the subsumption algorithm for the *set* of clauses D. We called them code tree for the following reasons:

1. They can be considered as a code for the subsumption algorithm;
2. They have a tree-like structure;
3. They can grow and shrink dynamically: new paths can be added and old paths can be pruned;

The use of the code tree is the following. With each new clause d to be added to the database D, we *integrate* the corresponding instruction sequence *matchspec*(d) into the code tree T. The integration process consists of finding the longest path in the tree which coincides with *matchspec*(d) (disregarding forks). Then we add a new fork together with the rest of *matchspec*(d) at the node where the path must diverge with *matchspec*(d). We denote the insertion operation $Insert(I, T)$.

We also use the procedure of *removing* the path *matchspec*(d) from the tree. We need to remove it for efficiency reasons when d is removed from the database (for example by back subsumption). The removal consists of finding the path that corresponds to *matchspec*(d) and removing the part of this path starting from the last fork together with the fork. We denote the removal operation $Remove(I, T)$.

When we add/remove instruction sequences we have to also change some arguments of instructions in the code tree, as can be seen on Figure 5. Assume that the tree consists of the instructions for the first clause $P(x, a, x)$ and we merge the instructions for the second clause $P(x, x, b)$ into it. Then the second argument to

[9] We have organized the abstract machine instructions in such a way that the only value which needs to be restored is the position of the head, i.e. the value of q. We achieved this by making *subst*, *post* arrays instead of stacks which resulted in having additional arguments to *Push*, *Pop* and *Put*.

the *Right* instruction must be changed from the address of *Check a* to the address of *Fork*.

Now we shall explain the semantics of the instruction set used in code trees similar to the one given on Figure 3. First of all we note that all previously used instructions have an additional argument representing the next instruction. After having executed an instruction, we have to pass the control to this next instruction. For example, *Right n* now means

$q := right(q)$;
`goto` n

The other instructions are changed in the same way. To implement new instructions *Fork* and *Restore* we introduce two new stacks *back* and *heads* representing backtrack points and head positions that need to be restored. Each *Fork* instruction creates a backtrack point and memorizes the current position of the head q. Each *Restore* instruction restores the position of q. Instructions *Check* and *Compare* which may fail now try to backtrack to the previously stored backtrack point:

`if` $var(q)$ `or` $functor(q) \neq f$
`then` `if` $empty(back)$
    `then` exit with fail
    `else` `goto` $pop(back)$
`else` `goto` i_k
...
	Check $f\ i_k$	$push(q, heads)$;
		$push(i_j, back)$;
		`goto` i_k;
		...
			Fork $i_j\ i_k$

		$q := pop(heads)$; `goto` i_k; ...	*Restore* i_k

We defined the semantics of the code tree instructions. The formal proof of correctness may be found in [Vor94]. In [Vor94] it is also defined how a code tree *corresponds* to a set of clauses:

Theorem 3. *Let the code tree T correspond to the set of clauses C and d be a clause. Then the execution of T on d terminates with success iff d is subsumed by a clause in C.*

Theorem 4. *Let the code tree T correspond to a set of clauses. Let n be the number of nodes in T. Then, for every clause c, execution of T on c calls at most n instructions.*

This theorem formally states the importance of having less instructions in a code tree. It is thus important to have more *shared instructions* in the tree, i.e. instructions that lie on many paths. The need for more sharing explains some features on the design of code trees. In many cases we could speed up single instruction sequences for subsumption by changing the order of instructions or by introducing new instructions. But it would create code trees with less sharing and thus decrease the efficiency of the "set at a time" subsumption.

Note that by using code trees we obtained an interesting technique of indexing in presence of variables. It has been noted in [SR93] that the non-ground subsumption-checking is difficult: "In the general case, subsumption-checking is a costly operation, and we are not aware of efficient subsumption-checking techniques for the case of arbitrary non-ground facts...". Variables create a problem for indexing because in most (all?) logic programming and theorem proving implementations they are implemented as addresses in memory. Any two occurrences of variables in two different clauses are two *different* addresses. Our technique discriminate variables in the order of their occurrences in clauses. Thus, it *treats variables occurring in different clauses* *in a uniform way.* We do not know any implementation of a logic programming or theorem proving system which makes use of this idea. In [Gra92] a technique is proposed which tries to find some dependencies in path indexes in a uniform way. Ideas of that paper have something in common with our paper.

There are some quite obvious optimizations of code trees. For example, one can use hashing instead of forking in the tree. When the set of function/predicate symbols is fixed in advance, one can even implement parts of code trees via arrays. Another possibility is to introduce explicit backtrack points in the trees, instead of putting them in a stack at run time. It makes the insertion and removal operations on trees more costly, but it can speed up the evaluation in general. In deductive database applications, where tuples are usually implemented via records, one can consider an alternative to $Down$ and $Right$ instructions. It is also possible to access arguments of functions in a non-standard order, depending on their types. When the main memory is insufficient to keep the whole code trees, we can organize them in such a way that they will load/save their parts automatically and keep information about subtree sizes etc. In general when using code trees, one can try to get the best both of the particular properties of data and the properties of the computer.

There is a little overhead on performing operations involved in the use of code trees, but they are all quite inexpensive. Compiling a term t into an instruction sequence is a cheap operation. Its cost is about the same as the subsumption-check for t by t. The complexity of this operation depends on the representation of variables, and can be made linear of the size of t. The merge and the removal of an instruction sequence into the tree are also quite cheap. The most difficult part of it is the finding of the longest path in the tree corresponding to t. It depends on the branching factor m of the tree (the number of forks at a fork node) and the size n of t, and is $O(m \times n)$ in the worst case.

4 Multi-literal clauses

According to our terminology, a multi-literal clause is any clause that is not a fact. Thus, Horn clauses which are not facts, are multi-literal. Disjunctive clauses, used in disjunctive logic programs, are also multi-literal. Clauses used in theorem proving are very often multi-literal.

We do not have enough space to explain all details of the multi-literal case. We shall only briefly overview the differences introduced by code trees for multi-literal clauses.

Assume that we need to make the subsumption-check for clauses c, d. Then every literal in c must subsumes a literal in d, and the substitution must be the same for all literals in c. It makes subsumption check much harder, since every literal in d can be selected as a potential candidate for a literal in c.

The multiple-literal subsumption can also be specialized. All commands used in the abstract subsumption machine for unit clause are needed for multi-literal clauses. In addition, we need two new instructions. The first instruction, $First$ sets the first literal in d as the current potential candidate to be subsumed by a literal in c. The second instruction $Next$, resets this candidate to be the next literal. Both instructions create new backtrack points: $Next$ can be executed as many times as the number of literals in d, which is unknown at the time when we compile the specialized procedure $subsume_c$. Thus, both commands have an additional argument which denotes the backtrack point, namely the (address of) the $Next$ instruction. Since c can also be multi-literal we need to remember the current candidates for each literal in c. To this end we introduce an array Q instead of the head q. Now q will denote the number of the current element in Q and $Q[q]$ can be considered as the head of the abstract machine. The semantics of the *Initialize* instruction changes. The new instructions of our machine have the following semantics:

```
q := -1;
goto j                  Initialize j
...
q := q + 1;
Q[q] := d;
Push(i, back)
goto j                  First i j
...

if defined(right(Q[q]))
then
{
  Q[q] := right(Q[q]);
  Push(i, back)
}
else
{
  q := q - 1;
  if empty(back)
  then exit with fail
  else goto pop(back)
}
goto j                  Next i j
...
```

The specializing algorithm for multi-literal clauses is very similar to the one for unit clauses. It compiles all literals one by one and adds two instructions *First* and *Next* before the code for every literal. The merge of instruction sequences into code trees is also very similar to the one for unit clauses. For example, the instruction sequences for clauses $P(a,x), P(x,b)$ and $P(a,y), P(f(y),y)$ and the corresponding code tree are given in Figure 6. As one can see, the code tree for this example shares the code for the first literals in both clauses and a part of the code for the second literal.

The code tree in this example is, strictly speaking, not a tree any more. The execution of the code tree is not the traversal of a relevant part of the tree. Now some parts of the code tree may be executed several times upon backtracking caused by the execution of *Next* instructions. It means that Theorem 4 about the number of steps called by the execution of a code tree is not true for multi-literal clauses. All other statements from the previous sections remain true for multi-literal clauses.

A simple optimization for Horn clauses is possible. Since there is only one positive literal in a Horn clause, and a positive literal can only be subsumed by a positive literal, no backtrack is needed for positive literals. It means that it is better to put the instruction sequence for the positive literal first, and the instructions *Next* for the positive literal is not necessary any more.

In our experiments with multi-literal clauses we had examples when the speedup of interpreted code trees over the OTTER [McC92] theorem prover on forward subsumption was over the factor of 30. (OTTER is considered as a standard for an efficient implementation).

5 Comparison with indexing

A number of approaches have been proposed to improve the performance of subsumption and other algorithms used in bottom-up procedures. Some of them are reviewed in [McC92]. Most of these techniques use indexing to filter out the relevant clauses and to perform the algorithms “set at a time”. There are two main types of indexing schemes which are used in modern theorem provers — path indexing

i_0: *Initialize* i_1
i_1: *First* i_2 i_3
i_2: *Next* i_2 i_3
i_3: *Check* P i_4
i_4: *Down* i_5
i_5: *Check* a i_6
i_6: *Right* i_7
i_7: *Put* 0 i_8
i_8: *First* i_9 i_{10}
i_9: *Next* i_9 i_{10}
i_{10}: *Check* P i_{11}
i_{11}: *Down* i_{12}
i_{12}: *Put* 1 i_{13}
i_{13}: *Right* i_{14}
i_{14}: *Check* b i_{15}
i_{15}: *Compare* 0 1 i_{16}
i_{16}: *Success*

j_0: *Initialize* j_1
j_1: *First* j_2 j_3
j_2: *Next* j_2 j_3
j_3: *Check* P j_4
j_4: *Down* j_5
j_5: *Check* a j_6
j_6: *Right* j_7
j_7: *Put* 0 j_8
j_8: *First* j_9 j_{10}
j_9: *Next* j_9 j_{10}
j_{10}: *Check* P j_{11}
j_{11}: *Down* j_{12}
j_{12}: *Check* f j_{13}
j_{13}: *Push* 0 j_{14}
j_{14}: *Down* j_{15}
j_{15}: *Put* 1 j_{16}
j_{16}: *Pop* 0 j_{17}
j_{17}: *Put* 2 j_{18}
j_{18}: *Compare* 0 1 j_{19}
j_{19}: *Compare* 0 2 j_{20}
j_{20}: *Success*

$\longrightarrow$

i_0: *Initialize* i_1
i_1: *First* i_2 i_3
i_2: *Next* i_2 i_3
i_3: *Check* P i_4
i_4: *Down* i_5
i_5: *Check* a i_6
i_6: *Right* i_7
i_7: *Put* 0 i_8
i_8: *First* i_9 i_{10}
i_9: *Next* i_9 i_{10}
i_{10}: *Check* P i_{11}
i_{11}: *Down* k_0

k_0: *Fork* k_1 i_{12}
i_{12}: *Put* 1 i_{13}
i_{13}: *Right* i_{14}
i_{14}: *Check* b i_{15}
i_{15}: *Compare* 0 1 i_{16}
i_{16}: *Success*

k_1: *Restore* j_{12}
j_{12}: *Check* f j_{13}
j_{13}: *Push* 0 j_{14}
j_{14}: *Down* j_{15}
j_{15}: *Put* 1 j_{16}
j_{16}: *Pop* 0 j_{17}
j_{17}: *Put* 2 j_{18}
j_{18}: *Compare* 0 1 j_{19}
j_{19}: *Compare* 0 2 j_{20}
j_{20}: *Success*

Fig. 6. Instruction sequences for multi-literal clauses $P(a,x), P(x,b)$ and $P(a,y), P(f(y),y)$ and the corresponding code tree.

[Sti89] and discrimination trees [McC92]. In this section we briefly compare code trees with indexing schemes.

Discrimination trees are trees which encode structure of terms occurring in clauses. Paths is the discrimination trees correspond to the terms. An example from [McC92] of a set of terms and its discrimination tree is shown on Figure 7 (the example is taken over from [McC92]).

1. $f(x,x)$
2. $f(x,y)$
3. $f(x,b)$
4. $f(g(a),x)$
5. $f(g(a),b)$
6. $f(a,y)$
7. $g(x)$
8. $g(z)$
9. $g(b)$

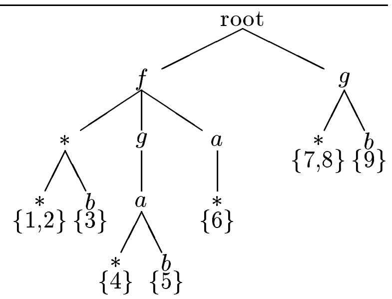

Fig. 7. An example of a set of terms and its discrimination tree

The main difference in the design of code trees and indexing schemes is the following. Indexing is used to filter out relevant clauses for a procedure. It may often be not enough for efficient algorithms. Thus, path indexes and discrimination trees are often provided with additional information. For example, in OTTER leaves of the discrimination tree contain not only references to clauses, but also the lists of variables in clauses. It sometimes helps to complete an operation. e.g., subsumption, without actually doing subsumption. The traversal (or, better, execution) of code trees is a complete algorithm. Code trees use a different idea: instead of indexing on data they dynamically create the *code* for a procedure. In the case of forward subsumption, where the structure of specialized procedures is very close to the structure of data, code trees and discrimination trees are very similar. However the code trees for back subsumption are radically different (we did not have enough space to consider them). The experience shows that discrimination trees give almost no speedup for back subsumption. One of the reasons for it lies in the fact that the code for back subsumption does not so closely resemble the structure of the kept clause as the code for forward subsumption. In the case of back subsumption, the specialized instructions for a clause already form a tree. Then these trees are merged into a big code tree. It is trivial to restore the database by its forward subsumption code tree. It is very difficult to do it for the back subsumption code tree.

It is also very important that code trees can be translated into the native code of a computer. We made experiments with compiled into the native code specialized algorithms. The real compilation can give another speedup of the factor of 3-4.

The idea of indexing in the case of subsumption can be summarized as follows: find structural properties P_1, P_2 of data such that whenever c_1 satisfies P_1 and t_2 satisfies P_2, then c_1 subsumes (or does not subsume) c_2. Then keep information about these P_1, P_2 in special data structures, for example discrimination trees. We propose to exploit common properties of code, which is not the same as indexing. In general, code trees are not just compiled discrimination trees. We do not claim that the use of code trees is always superior over indexing. They are just another technique to implement a class of procedures.

We have already seen how the use of code instead of data helped us to find a new way of indexing on variables. The same considerations resulted in an indexing scheme for multi-literal clauses. It shows that the decision to manipulate with code instead of data helps to find new design decisions. *Code trees provide an important methodology for designing bottom-up procedures. The consistent use of this methodology helps to find new solutions to old problems.* We believe that code trees give a different viewpoint on problems and their solutions, compared to the idea of indexing data.

Discrimination trees and other indexing methods require less memory than code trees, when several procedures participate in deduction. In theorem proving applications, some problems use resolution, both kinds of subsumption, rewriting and paramodulation. The full use of code trees would require 5 different trees to be constructed. But all these procedures may use the same indexing scheme. At the same time if one would try to provide indexing schemes with more information helpful for an algorithm, it may result in a huge memory consumption, as was shown in [Gra92] for path indexing.

6 Examples

Here we give a statistics of two examples proved in our theorem proving system `Vampire` on an 486 IBM PC (66Mh) running under Windows.[10].

[10] The system `Vampire` is now also available for Sun/Sparc and Hewlett-Packard computers

One of the hardest problems dealing with *unit clauses* was the condensed detachment 5 problem from [WOL91]. The initial clauses are the following:

1. $P(x_1) \vee \neg P(x_2) \vee \neg P(i(x_2, x_1))$.
2. $\neg P(i(i(a, b), i(i(b, c), i(a, c))))$.
3. $P(i(x_1, i(i(x_2, i(x_3, x_1)), i(x_3, x_2))))$.

The problem has been tried by the positive hyperresolution. In this example 21,262 clauses were kept, 4,449,518 clauses forward subsumed. The time spent by forward subsumption is 1959 seconds. It is difficult to say how many forward subsumption operations would have been be performed if subsumption has been implemented "clause at a time". If we take it to be 1/4 of the worst case $21,262 \times 4,449,518$, then the number of atomic subsumption operations would be $23,651,412,929$ which corresponds to more then $12,000,000$ subsumption-checks per second. The same computer can perform less than $2,000,000$ calls of "empty function" per second. A typical example of a clause from the proof is

$$P(i(i(i(i(i(x_1, x_2), i(x_2, x_3)), i(x_4, i(x_2, x_3))), x_5), i(x_6, x_5))).$$

Subsumption is very hard for such kind of problems for the following two reasons:

1. The number of variables in clauses is quite big (usually 7 to 10 in this example).
2. There is only one predicate symbol and only one function symbol. Thus, subsumption never fails on the comparison of function symbols.

On *multi-literal clauses,* we tried the "steamroller" problem from the Pelletier list [Pel86] by the "brute force" binary resolution (no restrictions or optimizations). The number of kept clauses in the final proof was 8,845, the number of forward subsumed clauses 50,242. The forward subsumption time was 138 seconds. Using the same estimations as above, we obtain that the performance corresponds to more than 800,000 subsumption-checks per second. In this example the subsumption check was extremely hard. There were no very deep terms, the main source of problems was the number of literals in clauses. The longest kept clause had 15 literals. (Subsumption on multi-literal clauses is NP-complete.) The average number of literals in clauses was about 7 (though there were both very short and very long clauses). Typical examples of clauses used in the final proof are:

5. $R(x_1, f_0(x_1)) \vee \neg P_5(x_1)$.
14. $R(x_1, x_2) \vee R(x_1, x_3) \vee \neg Q_0(x_3) \vee \neg R(x_2, x_4) \vee \neg Q_0(x_4) \vee \neg S(x_2, x_1) \vee \neg P_0(x_2) \vee \neg P_0(x_1)$.
1277. $\neg P_0(c_5) \vee \neg P_0(c_4) \vee \neg Q_0(x_1) \vee \neg R(c_4, x_1) \vee \neg Q_0(x_2) \vee R(c_5, x_2) \vee R(c_5, c_4)$.

On this example the indexing methods used in other theorem provers are inadequate. For example, discrimination trees as they are used in OTTER are of almost no use when they encounter terms like $P_5(x_1)$. Path indexing, even extended, does not work well on shallow terms and multi-literal clauses [Gra92].

Acknowledgments. Discussions with Micha Meier and Mark Wallace were the most stimulating. A visit to Argonne National Laboratory and discussions with Ewing Lusk, Bill McCune, Ross Overbeek and Larry Wos helped me to understand problems arising in handling large amounts of clauses. The motivation to efficiently implement some basic procedures are mostly due to the study of the excellent performance of OTTER implemented by Bill McCune. Comments on OLDT-resolution compilation from David S. Warren and his team were also very informative for me.

References

[AS92] O.L. Astrakhan and M.E. Stickel. Caching and lemmaizing in model elimination theorem prover. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 224–239, Saratoga Springs, NY, USA, June 1992. Springer Verlag.

[BGL93] R. Barbuti, R Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, Cambridge, MA, March 1986.

[BNRST87] C. Beeri, Sh. Naqvi, R. Ramakrishnan, O. Shmueli, and Sh. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 21–36. ACM Press, 1987.

[BR87] C. Beeri and R. Ramakrishnan. On the power of Magic. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283. ACM Press, 1987.

[BM72] R.S. Boyer and J.S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 101–116. Edinburgh University Press, Edinburgh, 1972.

[CD93] M. Codish and B. Demoen. Analyzing logic programs using "Prop"-ositional logic programs and a magic wand. In Dale Miller, editor, *Logic Programming — Proceedings of the 1993 International Symposium*, pages 114–129. The MIT Press, 1993.

[Das92] S.K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley, 1992.

[Dec86] H. Decker. Integrity enforcements on deductive databases. In *Proc. of the 1st International Conference on Expert Database Systems*, pages 271–285, Charleston, South Carolina, April 1986.

[Gra92] P. Graf. Path indexing for term retrieval. Technical Report MPI-I-92-237, Max-Planck Institut für Informatik, Saarbrücken, Germany, December 1992.

[LST86] J.W. Lloyd, E.A. Sonenberg, and R.W. Topor. Integrity constraint checking in stratified databases. Technical Report 86/5, Department of Computer Science, University of Melbourne, 1986.

[LT85] J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2(2):93–109, 1985.

[MB88] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. In *CADE'88 (9th Int. Conf. on Automated Deduction)*, Lecture Notes in Computer Science, pages 179–216, Argonne, Illinois, May 1988.

[McC88] William W. McCune. An indexing method for finding more general formulas. *Association for Automated Reasoning Newsletter*, 1(9):7–8, 1988.

[McC90] William W. McCune. OTTER 2.0 users guide. Technical report, Argonne National Laboratory, March 1990.

[McC92] William W. McCune. Experiments with discrimination-tree in indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Mas67] S.Yu. Maslov. An inverse method for establishing deducibility of nonprenex formulas of the predicate calculus. In J.Siekmann and G.Wrightson, editors, *Automation of Reasoning (Classical papers on Computational Logic)*, volume 2, pages 48–54. Springer Verlag, 1983.

[Nei90] V. Neiman. Refutation search for horn sets by a subgoal-extraction method. *Journal of Logic Programming*, 9(2):267–284, 1990.

[Pel86] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[Rob65a] J.A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[RW69] G. Robinson and L.T. Wos. Paramodulation and theorem-proving in first order theories with equality. In *Machine Intelligence*, volume 4. Edinburgh University Press, Edinburgh, 1969.

[Sti88] M. Stickel. A PROLOG technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, (4):353–380, 1988.

[Sti89] M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.

[SR93] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms (extended abstract). In Dale Miller, editor, *Logic Programming. Proceedings of the 1993 International Symposium*, pages 557–574. The MIT Press, 1993.

[TS86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[Vie89] Laurent Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[Vor90] A. Voronkov. LISS - the Logic Inference Search System. In Mark Stickel, editor, *Proc. Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 677–678, Kaiserslautern, Germany, 1990. Springer Verlag.

[Vor92] A. Voronkov. Theorem proving in non-standard logics based on the inverse method. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 648–662, Saratoga Springs, NY, USA, June 1992. Springer Verlag.

[Vor94] A. Voronkov. The Anatomy of Vampire: Implementing Bottom-up Procedures with Code Trees. *Submitted to Journal of Automated Reasoning*, 1994.

[War83] David H. D. Warren. An abstract Prolog instruction set. SRI Tech. Note 309, SRI Intl., Menlo Park, Calif., 1983.

[War92] D.S. Warren. Memoing for logic programs. *Communications of the ACM (invited paper)*, 35(3):93–111, 1992.

[WOL91] Larry Wos, Ross Overbeek, and Ewing Lusk. Subsumption, a sometimes undervalued procedure. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson.*, pages 3–40. The MIT Press, Cambridge, Massachusetts, 1991.

[Wos92] Larry Wos. Note on McCune's article on discrimination trees. *Journal of Automated Reasoning*, 9(2):145–146, 1992.