UPMAIL Technical Report No. 88
October 3, 1994

Implementing Bottom-up Procedures with Code Trees: a Case Study of Forward Subsumption

Andrei Voronkov[1]

Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala,
Sweden

email voronkov@csd.uu.se

[1]Supported by Swedish TFR grant no. 93-409

Abstract

We present an implementation technique for a class of bottom-up logic procedures. The technique is based on *code trees.* It is intended to speed up most important and costly operations, such as subsumption and resolution. As a case study, 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 makes subsumption-check 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 binary resolution, hyperresolution, UR-resolution, the inverse method, paramodulation and rewriting, OLDT-resolution, SLD-AL-resolution, bottom-up evaluation of logic programs and disjunctive logic programs.

Contents

List of Figures

Section 1

Introduction

Most of automatic theorem proving systems can be generally divided in two parts by their mode of evaluation (or clause generation). *Top-down* systems start from a goal, reducing it to subgoals, until all subgoals become axioms. Such systems usually exploit just one branch of the search tree. *Bottom-up* systems usually deal with a database of clauses and generate new clauses by applying inference rules to clauses in the database. There are also systems which can combine top-down with bottom-up.

Systems which keep track of previously generated clauses are said to *retain information* in [WOL91]. All bottom-up systems retain information. If a top-down system retains information, it is usually characterized as a combination of top-down and bottom-up.

The systems retaining information show a superior performance when dealing with problems involving large search spaces. It has been widely recognized in the deductive database community. There are two main reasons why systems that retain information become more efficient on hard problems:

1. Systems that retain no information must often do the same job on different branches of the search tree.

2. In systems that retain information one can implement major operations on the "set at a time" basis, opposite to "tuple a time" or "clause at a time" procedures used by e.g. Prolog.

Procedures retaining information have been traditionally used in automated deduction, since its very beginning. Recently, the need for such procedures has been recognized in logic programming and deductive databases. In logic programming, it has been noted that tabulation can both speed up evaluation and give more declarative treatment of some subgoals. In deductive databases, where the databases can be very large, the old "tuple at a time" methods are not adequate any more. Also, deductive databases do not allow the query answering process be controlled by a user. In this case recursive specifications can cause non-termination, unless (some) intermediate goals are retained. In logic programming and deductive databases the systems that make use of previously generated clauses (or goals) are usually considered as bottom-up procedures. We shall use "bottom-up" to denote procedures that retain clauses throughout this article.[1]

The necessity to retain information leads to the introduction of bottom-up features into top-down systems in various areas around automated deduction. For instance, various modifications of magic sets transformations require the semi-naive bottom-up evaluation of the transformed logic

[1] The bottom-up methods of search are also sometimes called *indirect* methods, in opposite to *direct* top-down methods. (Lincoln Wallen, private communication.)

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. In automatic theorem proving, model elimination which is a top-down procedure, has recently been augmented with lemmas [AS92, GLMS94], which allow one to use previously proved subgoals.

This article deals with an implementation technique for bottom-up systems, which can be used to speed up many important algorithms.

Bottom-up algorithms have been used through the years in automated deduction. They 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.

Most bottom-up systems are based on the following loop that resembles the "closure algorithm" from [Lus92]:

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. Change 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.

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 changes in D are induced by new clauses from ΔD. Two algorithms may be applied at this stage. *Back subsumption* consists of removal from D every clause subsumed by a clause from ΔD. If a new clause includes equalities, it can be used for *back demodulation*, i.e. rewriting applied to clauses from D.

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], bottom-up static analysis of logic programs [CD93], constraint logic programming, parsing [TR94], production systems [MAT94]. Top-down procedures which use lemmaizing, like SLD-AL-resolution [Vie89] or some implementations of model elimination [AS92] also use a similar scheme.

There are certain specific features of such systems which pose a challenge to people who implement them:

1. There is usually a small number of operations which must repeatedly be applied to clauses in the database (for example, resolution).

2. The database may be very large and dynamically changing.

3. The number of calls of these operations is so big that it is practically impossible to implement them on the "clause at a time" basis.

Below we shall consider particular examples which illustrate the number of operation calls for some hard problems in automated deduction.

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 meet 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.[2]

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. The second idea of this paper is to perform subsumption on the "set at a time" basis via code trees which merge several specialized algorithms into one.

[2] In our experiments we tried benchmarks where around 900,000,000 clauses were generated, and the number of subsumption-checks was about 20,000,000,000.

Section 2

Subsumption. The abstract subsumption machine

We assume acquaintance with the basic notions of terms and substitutions. A *clause* is a list of terms. 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[1]. 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 clauses are used in disjunctive logic programs, theorem proving procedures and static analysis of logic programs. **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 2.1. This clause will be our running example until the end of Section 3.

In order to present our algorithms, we need to introduce a few operations on positions. Let p be a position, corresponding to t_i in a subterm $f(t_1, \ldots, t_n)$. Then $right(p)$ is the position to the right of p, i.e. the one that corresponds to t_{i+1}. If such position does not exist, $right(p)$ is undefined. For example, on Figure 2.1 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

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

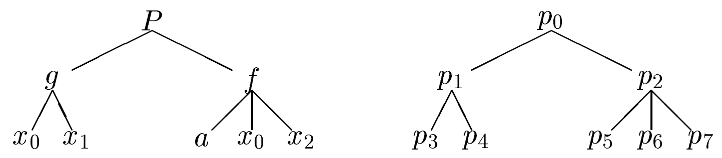

Figure 2.1: The clause $P(g(x_0, x_1), f(a, x_0, x_2))$ and its set of positions $\{p_0, \ldots, p_7\}$.

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 2.1 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[2]. $defined(p)$ 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 2.1 $var(p_0)$ is false, $var(p_7)$ is true, $functor(p_0) = P$, $p_3 == p_6$ is true and $p_3 == p_1$ is false.

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 2.2. 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* tries to construct a substitution θ such that $c_1\theta = c_2$. First of all, *match* traverses the c_1 and c_2, 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 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.*

Proof is omitted.

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 is 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 2.3.[3]

[2]In logic programming languages such use of functors is allowed. In this case we have to consider pairs g/n

```
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 defined(right(p)) then push(⟨right(p), right(q)⟩, post);
      if defined(down(p)) then push(⟨down(p), down(q)⟩, post)
      }
    }
  forall cliques c = ⟨p0, q0⟩, ..., ⟨pn, qn⟩ ∈ subst
  forall i ∈ {1, ..., n} {
    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 $j \in \{0, \ldots, k\}$.

Figure 2.2: The general matching algorithm for clauses c_1, c_2

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 g*
$post[0] := right(q)$;	*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	

Figure 2.3: 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 Roman letter, like Right.

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 2.3. 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 positions to be considered later and *subst* for keeping tracks of terms substituted for variables. The meaning of the set of instructions is the following:

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.

The specializing algorithm *matchspec* accepts a term c_1 as the input and constructs the instruction sequence for the subsumption of c_2 by c_1. The algorithm *matchspec* is shown in Figure 2.4. The following theorem shows the soundness and correctness of the of the *matchspec* algorithm:

Theorem 2 *For all unit clauses c_1, c_2, $matchspec(c_1)(c_2)$ is equivalent to $match(c_1, c_2)$, i.e. one of them terminates with success iff the other one terminates with success.*

Proof (sketch). By induction on the size of c_1 one can show the following. Let mat_{c_1} be obtained by the unfolding of *match* using c_1 as the first argument. Replace in mat_{c_1} all sequences of instructions of the form

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

where E is an arbitrary expression by the equivalent $q := E$. It is easy to see that we obtained exactly $matchspec_{c_1}$.

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

consisting of a function symbol and its arity. We do not treat this case for the sake of simplicity.

[3]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 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 defined(down(p))
    then {
      if defined(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 defined(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
```

Figure 2.4: The specializing algorithm $matchspec(c)$.

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.*

Section 3

Code trees

It has been widely recognized in the deductive database research that the logic programming "goal at a time"[1] 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.*[2] 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 3.1 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

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

[2] 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.

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

Figure 3.1: Two instruction sequences and the corresponding code tree. Shared parts of code are put in a dashed box.

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 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*.[3] The new set of instructions representing subsumption for this set of two clauses is shown on Figure 3.1 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 existing 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 3.1. Assume that the tree consists of the instructions

[3] 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 implementing stacks *subst*,*post* via arrays which resulted in having additional arguments to the *Push*, *Pop* and *Put* 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 argument of the *Right* instruction must be changed from the address of the *Check a* instruction to the address i_5 of *Fork*.

Now we shall explain the semantics of the instruction set used in code trees similar to the one given on Figure 2.3. 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. In order to prove that we use code trees correctly we introduce some formal definitions. We call *extended instructions* all previously introduced instruction having the additional argument representing the next instruction. A *code tree* is any finite set of extended instructions. We call arguments representing instructions *addresses* of these instructions ($i_0, \ldots, j_{10}$ on Figure 3.1. We shall often identify instructions in a code tree with their addresses. An *extended path* in the code tree is a any sequence of instructions defined by induction in the following way:

1. Any single instruction is an extended path.
2. Let $i_1, \ldots, i_n$ be an extended path.
   (a) If i_n has the instruction i as its last argument then $i_1, \ldots, i_n, i$ is an extended path.
   (b) If i_n is a *Fork* j k instruction, then $i_1, \ldots, i_n, j$ is an extended path.

Paths in the code tree are obtained from extended paths by deleting all *Fork* and *Restore* instructions and omitting the last argument of each remaining instruction, except for *Success*. For example, the extended path

i_1 : *Initialize* i_6		
i_6 : *Fork* i_7 i_8	gives the path	*Initialize*
i_8 : *Restore* i_{12}		*Check* P
i_{12} : *Check* P i_{16}		

A *completed (extended) path* is an (extended) path whose last instruction is *Success*. An instruction j is *reachable* from i in the code tree T iff there is an extended path $i, \ldots, j$ in T. A code tree T is *consistent* iff it satisfies the following properties:

1. There is exactly one *Initialize* instruction in T.

2. Every instruction $i \in T$ is reachable from the *Initialize* instruction.

3. For every instruction $i \in T$ which is not a *Success* instruction, there is a *Success* instruction reachable from I.

4. For every instruction $i \in T$, the last argument of i is not a *Restore* instruction.

5. For every *Fork* i j instruction, i is a *Restore* instruction.

6. There are no loops in T, i.e. the lengths of extended paths in T are restricted.

We shall only consider consistent code trees.

A code tree T *corresponds* to a set of clauses $c_1, \ldots, c_n$ iff the set of all paths in T is exactly $\{match_{c_1}, \ldots, match_{c_n}\}$. Now we can formally define the *execution* of the code trees T with respect to a clause d as the sequence of extended instructions, starting from the *Initialize* instruction, which would have been executed by T on d according to the given semantics of the extended instructions.

Theorem 3 *Let the consistent 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.*

Proof (sketch).

1. Assume that the execution succeeds. Consider the sequence I of instructions executed by T. If a backtrack happened in I (i.e. two instructions *Fork* i j and i : *Restore* k were executed, then the part of I between this two instructions can be safely removed. After removal of all backtrack points from I, we obtain an extended path J. Let J' be the path obtained from J by removing all *Fork* and *Restore* instructions. It is easy to see that the executions of J and J' give the same result, i.e. both of them succeed. We know that $J' = matchspec(c)$ for some $c \in C$. Applying Theorems 1 and 2 we obtain that d is subsumed by c.

2. Assume that the execution fails. Since any instruction $i \in T$ is reachable from the *Initialize*, there is at least one failing *Check* or *Compare* instruction on any path. Applying the same arguments as above, we obtain that no $c \in C$ subsumes d.

A consistent code tree T is in the *tree form* iff

1. The last arguments of any two instructions in T are different.

2. The first arguments m of any two different *Fork* m n instructions are different.

Theorem 4 *Let the code tree T be in the tree form with n instructions. Then, for every clause c, execution of T on c calls at most n instructions.*

Proof is trivial.

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.

The following theorem shows that we achieved a maximal sharing on some instructions:

Theorem 5 *Let $I_1 = matchspec(c_1)$ and $I_2 = matchspec(c_2)$. Let the first n instructions of I_1, I_2 coincide. Let the $n+1$th instruction of I_1, I_2 be Push m, Push n. Then $m = n$. The same holds for Pop and Put instructions.*

Proof by routine inspection of *matchspec*.

Theorem 6 *Let T be a consistent code tree in the tree form. Let $I = matchspec(c)$. The $Insert(I,T)$ and $Remove(I,T)$ are consistent code trees in the tree form. Let, in addition, T correspond to a set of clauses C. Then*

1. *If no clause in C is a variant of c then $Insert(I,T)$ corresponds to $C \cup c$, otherwise $Insert(I,T)$ corresponds to C;*
2. *$Remove(I,T)$ corresponds to C without all variants of c;*

Proof is straightforward, but tedious.

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 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 [Gra94] 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.

Let us analyze at which circumstances code trees are most efficient. The first necessary condition for creating code trees is to have a simple abstract machine with a small number of instructions for executing a single problem. The second condition is that the structure of code admit sharing. The third condition is that the execution of instructions on a single problem should either produce no side-effects, or to produce side-effects which are easily recoverable. In the case of matching, the only side effect was the change of the position of the head q, which we restored upon backtracking. Fortunately, all major procedures used in automated deduction satisfy these three properties in most cases.

Section 4

Multi-literal clauses

In this section we consider multi-literal clauses. The subsumption-check for multi-literal clauses is much harder than that of unit clauses. It is known that the subsumption-check for multi-literal clauses is NP-complete (see e.g. [GJ79]). The usual indexing methods used in theorem proving programs do not work well with multi-literal clauses. For example, in OTTER the clause $P(x), P(f(f(x)))$ will be selected as a potential candidate to subsume *any* clause in which the predicate symbol P occurs, since discrimination trees and path indexes in OTTER make indexing on literals, not clauses.

Assume that we need to make the subsumption-check for clauses c, d. Then every literal in c must subsume a literal in d, and the matching substitutions must be compatible for all literals in c. Now every literal in d can be selected as a potential candidate for a literal in c.

The multi-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
...	
i : `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
...
```

More informally, the *First i j* command sets the (position of the) first literal of d as the candidate to be subsumed by the current literal in c (literal number q). *First* also sets an additional backtrack point i, corresponding to a *Next* instruction. *Next i j* is executed upon backtracking. It tries to reset the candidate for the literal number q to the next literal in d. If this next literal exists, *Next i j* puts the address i of itself as the new backtrack point. Otherwise, it backtracks, resetting q to $q-1$, i.e. trying to reconsider the matching for the previous literal.

This modification of the subsumption abstract machine is friendly to the code trees idea. The reason is that we still have a small number of instructions with easily recoverable side-effects and good sharing properties.

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 4.1. 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. However in the multi-literal case they are not obvious, since code trees now may have infinite paths caused by repeated applications of the *Next* instruction.

Instructions, extended instructions and *code trees* are defined as in the previous section, but may now include *First i j* and *Next i j* instructions. An *extended path* is now defined in the following way:

1. Any single instruction is an extended path.
2. Let $i_1, \dots, i_n$ be an extended path.
   (a) If i_n has the instruction i as its last argument then $i_1, \dots, i_n, i$ is an extended path.
   (b) If i_n is a *Fork j k*, or a *First j k* or a *Next j k* instruction, then $i_1, \dots, i_n, j$ is an extended path.

i_0 : *Initialize* i_1	j_0 : *Initialize* j_1	i_0 : *Initialize*i_1	
i_1 : *First* i_2 i_3	j_1 : *First* j_2 j_3	i_1 : *First* i_2 i_3	
i_2 : *Next* i_2 i_3	j_2 : *Next* j_2 j_3	i_2 : *Next* i_2 i_3	
i_3 : *Check* P i_4	j_3 : *Check* P j_4	i_3 : *Check* P i_4	
i_4 : *Down* i_5	j_4 : *Down* j_5	i_4 : *Down* i_5	
i_5 : *Check* a i_6	j_5 : *Check* a j_6	i_5 : *Check* a i_6	
i_6 : *Right* i_7	j_6 : *Right* j_7	i_6 : *Right* i_7	
i_7 : *Put* 0 i_8	j_7 : *Put* 0 j_8	i_7 : *Put* 0 i_8	
i_8 : *First* i_9 i_{10}	j_8 : *First* j_9 j_{10}	i_8 : *First* i_9 i_{10}	
i_9 : *Next* i_9 i_{10}	j_9 : *Next* j_9 j_{10}	i_9 : *Next* i_9 i_{10}	
i_{10} : *Check* P i_{11}	j_{10} : *Check* P j_{11}	i_{10} : *Check* P i_{11}	
i_{11} : *Down* i_{12}	j_{11} : *Down* j_{12}	i_{11} : *Down* k_0	
i_{12} : *Put* 1 i_{13}	j_{12} : *Check* f j_{13}	k_0 : *Fork* k_1 i_{12}	k_1 : *Restore* j_{12}
i_{13} : *Right* i_{14}	j_{13} : *Push* 0 j_{14}	i_{12} : *Put* 1 i_{13}	j_{12} : *Check* f j_{13}
i_{14} : *Check* b i_{15}	j_{14} : *Down* j_{15}	i_{13} : *Right* i_{14}	j_{13} : *Push* 0 j_{14}
i_{15} : *Compare* 0 1 i_{16}	j_{15} : *Put* 1 j_{16}	i_{14} : *Check* b i_{15}	j_{14} : *Down* j_{15}
i_{16} : *Success*	j_{16} : *Pop* 0 j_{17}	i_{15} : *Compare* 0 1 i_{16}	j_{15} : *Put* 1 j_{16}
	j_{17} : *Put* 2 j_{18}	i_{16} : *Success*	j_{16} : *Pop* 0 j_{17}
	j_{18} : *Compare* 0 1 j_{19}		j_{17} : *Put* 2 j_{18}
	j_{19} : *Compare* 0 2 j_{20}		j_{18} : *Compare* 0 1 j_{19}
	j_{20} : *Success*		j_{19} : *Compare* 0 2 j_{20}
			j_{20} : *Success*

Figure 4.1: 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.

An extended k-path is an extended path in which the number of consecutive *Next* instructions is $\leq k$.

As before, a *completed* extended path is an extended path whose last instruction is *Success*. A code tree T is *consistent* iff it satisfies the following properties:

1. There is exactly one *Initialize* instruction in T.
2. Every instruction $i \in T$ is reachable from the *Initialize* instruction.
3. For every instruction $i \in T$ which is not a *Success* instruction, there is a *Success* instruction reachable from I.
4. For every instruction $i \in T$, the last argument of i is not a *Restore* or a *Next* instruction.
5. For every *Fork* i j instruction, i is a *Restore* instruction.
6. For every *First* i j instruction, i is a *Next* instruction.
7. For every *Next* i j instruction, i is (the address of) this instruction.
8. All loops in T are caused by repeated applications of *Next* instruction. More precisely, for every $k \geq 1$, the lengths of extended k-paths in T are restricted.

Let us denote by *subsume* and *subsumespec* the subsumption algorithm and the specialization algorithm for multi-literal clauses, which are constructed based on *match* and *matchspec*.

A code tree T *corresponds* to a set of clauses C iff the set of all 1-paths is exactly $\{subsumespec(c) \mid c \in C\}$.

Theorem 7 *Let the consistent 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.*

Proof (sketch)

($\Rightarrow$) Assume that the execution of T terminates with success. Consider the sequence of instructions $I = i_1, \ldots, i_n$ which have been followed during the execution. let J be obtained from I by removal of all backtrack points in the following way:

1. If there is an occurrence of two instructions *Fork* i j and i : *Restore* k , remove both instructions and all instruction in between.
2. If there is an occurrence of two instructions *First* i j and i : *Next* i k, remove all instruction between the two.

It is easy to see that the execution of J will have the same effect as the execution of I. Let P be the extended 1-path obtained from J in the following way: remove all multiple occurrences of the same *Next* instruction and add *Next* after each *First* not followed by *Next*. Thus, there is a clause $c \in C$ such that $subsumespec(c) = P$. Consider the substitution θ obtained by the execution of I. Let $f : N \to N$ be a partial function of natural numbers defined as follows: if the kth *First* instruction in J is immediately followed by n *Next* instructions, let $f(k) = n+1$. Using the properties of the *matchspec* algorithm and properties of *First*, *Next*, one can prove $c_k\theta = d_{f(k)}$, where c_j (d_j) denotes the jth literal of the clause c (d). Thus, c subsumes d.

($\Leftarrow$) Similarly. Let $c \in C$ subsumes d such that there is a substitution θ with $c_i\theta = d_{f(i)}$. Consider the extended path in T defined as previously. One can check that either the execution of T on d will follow this path and terminate with success, or there is path terminated with success before.

It is not possible any more to prove the theorem on the number of steps in the code tree execution, since instructions *Next* can cause a backtracking which causes the exponential upper bound on the number of steps. It is not surprising, since subsumption on multi-literal clauses is NP-complete. The exponent arises when we have (failed) *Next* instructions inside other *Next* instructions. Nevertheless, code trees show superior behavior over other approaches on multi-literal clauses for the following reason: such deep backtrack points may be merged together in code trees, thus avoiding re-executing them many times.

Theorem 5 about sharing of code in code trees remains valid for multi-literal code trees. Insertion/removal of code in the code tree for the multi-literal case are similar to those of the trees for matching. As in the unit case, both operations applied to consistent code trees give consistent code trees and Theorem 6 also holds in the multi-literal case.

Section 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 [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 5.1 (the example is taken over from [McC92]).

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, indexing schemes are usually 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 and unification are very different from those of forward subsumption. The

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. $g(b)$

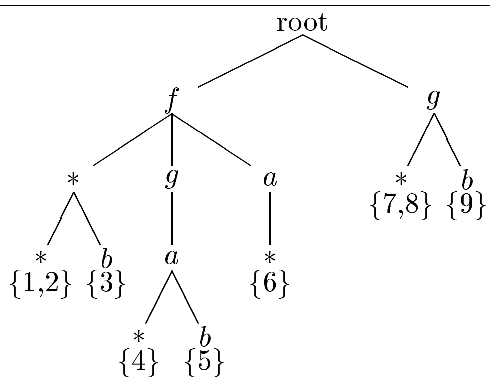

Figure 5.1: An example of a set of terms and its discrimination tree

experience shows that discrimination trees do not perform well 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 unification, the specialized instructions for a clause already form a tree with forks created by if-then-else statements.

It is also very important that code trees can be translated into the native code of a computer. We made experiments with specialized algorithms compiled into the native code. 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 shared code, which is not the same as indexing. 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 use of code trees also resulted in a new soluiton for subsumption on 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.

Indexing methods may 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 [Gra94] for path indexing.

Section 6

Experiments

Here we give a statistics of two examples proved in our theorem proving system *Vampire* on a Hewlett Packard 735. We did not try to set the best methods/switches to solve the two problems. On the contrary, we tried to set options in such a way so that to increase the number of forward subsumptions required.

One of the hardest problems dealing with *unit clauses* was the condensed detachment 4 problem from [WOL91]. The initial clauses are the following:

1. $P(x_1), \neg P(x_2), \neg P(i(x_2, x_1))$.
2. $\neg P(i(i(i(a, i(b, c)), c), i(b, a)))$.
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 positive hyperresolution with the restriction 20 on the maximal weight of clauses.[1] On this problem positive hyperresolution generates only unit clauses. A typical example of a clause used in this 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]:

1. $\neg Q1(x0), \neg P0(x1), \neg P0(x2), \neg R(x1, x0), \neg R(x2, x1)$
2. $P0(x0), \neg P1(x0)$
3. $P1(a)$
4. $\neg P2(x0), P0(x0)$
5. $P2(b)$
6. $\neg P3(x0), P0(x0)$
7. $P3(c)$

[1] Here the weight of a clause is the number of occurrences of symbols in the clause. For example, the weight of $P(f(x)), Q(a)$ is 5.

8. $\neg P4(x0), P0(x0)$
9. $P4(d)$
10. $\neg P5(x0), P0(x0)$
11. $P5(e)$
12. $Q0(x0), \neg Q1(x0)$
13. $Q1(g)$
14. $\neg Q0(x0), \neg Q0(x1), \neg P0(x2), \neg P0(x3), \neg S(x2, x3), R(x3, x2), R(x3, x1), \neg R(x2, x0)$
15. $\neg P5(x0), \neg P3(x1), S(x0, x1)$
16. $\neg P4(x0), \neg P3(x1), S(x0, x1)$
17. $\neg P3(x0), \neg P2(x1), S(x0, x1)$
18. $\neg P2(x0), \neg P1(x1), S(x0, x1)$
19. $\neg P2(x0), \neg P1(x1), \neg R(x1, x0)$
20. $\neg Q1(x0), \neg P1(x1), \neg R(x1, x0)$
21. $\neg P4(x0), \neg P3(x1), R(x1, x0)$
22. $\neg P5(x0), \neg P3(x1), \neg R(x1, x0)$
23. $Q0(f(x0)), \neg P4(x0)$
24. $\neg P4(x0), R(x0, f(x0))$
25. $Q0(f(x0)), \neg P5(x0)$
26. $\neg P5(x0), R(x0, f(x0))$

The problem has been tried by the binary resolution (no restrictions or optimizations). In order to increase the database size and the number of forward subsumptions, we switched back subsumption off. This problem have been tried with different weight restrictions. Typical clauses used in the found proof are the following:

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

There are no deep terms involved in the proof search, as in the first problem. But subsumption-checks for this problem are even more difficult for the following reason. This problem generates only a very small number of relatively short clauses. Most of clauses have from 7 to 11 literals. (The algorithm *subsume* on clauses with m and n literals, must perform m^n matches in the worst case.)

We used a counter which counted the number of subsumption-checks which would have been executed if subsumption had been implemented on the clause-to-clause basis. The results are given in Figure 6.1 on page 28:

As one can see from this table, Vampire achieved the speed of more than 200,000,000 subsumptions per second on unit clauses, where code sharing is very high. The speed is still very high (about 4,000,000 subsumptions per second) on multi-literal clauses with up to 15 literals.

We compare the performance of Vampire with that of Otter (Otter uses discrimination trees) on forward subsumption in Figure 6.2 on page 29.

In the case of unit clauses, the difference in performance is not high, since Otter uses a version of discrimination trees which encodes essentially the same information as code trees. The two provers use slightly different way of clause generation, so the number of kept/subsumed clauses is slightly different. In the case of the Steamroller problem, the binary resolution algorithms used by the

Problem/options	Kept clauses	Subsumed clauses	Clause-to-clause subsumptions	Forward subsumption time	Subsumptions per second	Database subsumptions per second
Condensed detachment (weight = 20)	30,509	497,141	1,581,472,615	20.14	78,523,963	26,199
Condensed detachment (weight = 24)	163,509	534,456	17,763,995,378	81.37	218,311,360	8,577
Steamroller (weight = 17)	12,161	872,061	2,861,295,196	355.53	8,059,986	2,491
Steamroller (weight = 20)	11,012	803,982	2,173,049,289	539.44	4,028,342	1,510
Steamroller (weight = 22)	26,185	2,074,607	9,695,480,345	2422.34	4,003,088	867
Steamroller (no restrictions)	41,743	3,048,078	17,133,045,644	4287.16	3,996,362	711

Note. For the weight = 24 case a proof was not obtained due to insufficient memory. The figures in the table are given up to the moment when Vampire ran out of memory after using about 62 megabytes

Figure 6.1: Experiments with subsumption

	Typical clause	Kept	Forward subsumed	Forward subsumption time	Ratio
	Condensed detachment (maximal weight = 20)				
Vampire Otter	$P(i(i(i(i(i(x_1,x_2),i(x_2,x_3)),$ $i(x_4,i(x_2,x_3))),x_5),i(x_6,x_5)))$	30,509 28,390	497,141 482,119	20.14 46.04	2.36–2.53
	Steamroller (maximal weight = 17)				
Vampire Otter	$\neg Q0(x0), \neg Q0(x1), \neg P2(g), \neg P0(g),$ $\neg S(g,g), R(g,x1), \neg R(g,x0)$	12,161 9,091	872,061 56,579	404 453	17.3–23.1
	Steamroller (maximal weight = 20)				
Vampire Otter	$\neg Q0(x0), \neg Q0(x1), \neg Q1(x2), \neg P4(x2), \neg P3(x1),$ $\neg P0(x3), \neg P0(x1), \neg S(x1,x3), \neg R(x1,x0)$	11,012 10,384	803,982 57,752	539 687	17.7–18.8
	Steamroller (maximal weight = 22)				
Vampire Otter	$\neg Q0(x0), \neg Q0(x1), \neg Q1(x2), \neg P4(x2), \neg P3(x1), \neg P0(x3),$ $\neg P0(x1), \neg P1(x3), \neg S(x1,x3), \neg R(x1,x0)$	26,185 11,397	2,074,607 58,567	2422 1,052	15.4–35.3
	Steamroller (no restrictions)				
Vampire Otter	$\neg Q0(x0), \neg Q0(x1), \neg Q1(x2), \neg P5(x3), \neg P3(x2), \neg P2(x2),$ $\neg P0(f6(x3)), \neg P0(x2), \neg S(f(x3),x2), R(x2,x1), \neg R(f6(x3),x0)$	41,743 15,949	3,048,078 63,970	4287 4430	49.2–128.9

Note. The minimal ratio in this table is based on the number of subsumed clauses per second, the maximal ratio takes into account the size of the database (i.e. the number of kept clauses).

Figure 6.2: Comparison of forward subsumption in Otter and Vampire

two provers were very different, which resulted in very different numbers of subsumed clauses. the difference is mostly due to th treatment of factoring in Vampire and Otter. However, the average size of generated clauses was about the same, so the comparison of performance is fair.

Acknowledgments. I am grateful to all people who made a direct or indirect influence on this paper. These are Anatoli Degtyarev, Ewing Lusk, Bill McCune, Micha Meier, Ross Overbeek, Mark Wallace, David S. Warren and Larry Wos. The motivation to efficiently implement some basic procedures is mostly due to the study of the excellent performance of OTTER implemented by Bill McCune.

Bibliography

[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.

[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.

[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.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[GLMS94] C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent developments. In A. Bundy, editor, *Automated Deduction — CADE-12. 12th International Conference on Automated Deduction.*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 778–782, Nancy, France, June/July 1994.

[GL87] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, April 1987.

[Gra94] P. Graf. Extended path-indexing. In A. Bundy, editor, *Automated Deduction — CADE-12. 12th International Conference on Automated Deduction.*, volume 814 of

Lecture Notes in Artificial Intelligence, pages 514–528, Nancy, France, June/July 1994.

[Lus92] E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Logic Programming and Automated Reasoning. International Conference LPAR'92.*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 96–106, St.Petersburg, Russia, July 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.

[MAT94] A.M. Maeda, J.-I. Aoe, and H. Tomabechi. Signature-check based unification filter. *Software — Practice and Experience*, 24(7):603–622, 1994.

[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.

[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.

[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.

[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.

[TR94] M. Tambe and P.S. Rosenbloom. Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68:155–190, 1994.

[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.

[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.