[Ghel 86] S. Ghelfo, G. Levi, A partial evaluator for metaprograms in a multiple theories logic language, ESPRIT Project EPSILON Report (October 1986).

[Ghel 87] S. Ghelfo, G. Levi, G. Sardu, Un valutatore parziale per metaprogrammi in un linguaggio logico con teorie multiple, GULP '87, 1987.

[Hog 81] C.J. Hogger, Derivation of Logic Programs, JACM 29, N.2 (1981) pp. 372-392.

[Hog 84] C.J. Hogger, Introduction to Logic Programming, Academic Press 1984.

[Par 83] H. Partsch, R. Steinbruggen, Program Transformation Systems, ACM Computing Surveys 15, N.3 (1983) pp. 199-236.

[Sato 84] T. Sato, H. Tamaki, Transformational Logic Program Systhesis, Proc. Int'l Conf. on Fifth Gen. Computer Systems (1984) pp. 195-201.

[She 84] J. C. Shepherdson, Negation as Failure : a comparison of Clark's Completed Data Base and Reiter's Closed World Assumption, J. Logic Programming, 1, pp. 1-48 (1984).

[She 85] J. C. Shepherdson, Negation as Failure II , J. Logic Programming, 3, pp. 185-202 (1984).

[Tak 86] A. Takeuchi, K. Furukawa, Partial evaluation of Prolog programs and its application to meta programming, in H. J. Kugler (ed.): Information Processing 86, Dublin, Ireland, North-Holland, pp. 415-420, 1986.

[Tam 83] H. Tamaki, T. Sato, A Transformation System for Logic Programs which Preserves Equivalence, ICOT TR-018 (1983).

[Tam 84] H. Tamaki, T. Sato, Unfold / fold transformation ol logic programs , Proc. Sec. Int'l Logic Progr. Conf. Upssala, pp.127-138, 1984.

[van E 76] M.H. van Emden, R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, JACM 23, N.4 (1976) pp.733-742.

[Ven 84] R. Venken, A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimization, in T. O'Shea (ed.) : ECAI84, North-Holland, pp. 91-104, 1984.

[Vie 86] L. Vielle, Recursion in Deductive Databases : A DB-Complete Proof Procedure Based on SLD-Resolution, Technical Report Kb-15 (1986).

[Zan 87] C. Zaniolo, D. Saccà, Rule Rewriting Methods for Efficient Implementations of Horn Logic, Proc. CREAS, Austin, 1987.

Optimization of logic programs execution based on their static analysis

C. Codognet[1] MM. Corsini[2] G. Filé[3]

Abstract A static analysis of pure PROLOG programs is used as the common basis of different techniques for optimizing the execution of these programs. We describe how the information computed during the static analysis of a program is useful for executing it in an AND-parallel fashion, performing intelligent backtracking, and in a space efficient way. Each of the techniques can be applied alone as well as together with the other ones.

Introduction

Improving the efficiency of the execution of pure PROLOG programs (simply programs in what follows) is our goal. To this end we propose to use some information about any given program that is computed by a static analysis of the program. This type of information is often called an abstract interpretation of the program, see [Son86,Mel86], where abstract interpretations are applied to the occur-check problem and the construction of modes . We go further, showing that this information can be used as the base of several different techniques for optimizing the execution of programs.

Roughly, the static analysis consists of computing, for each literal l of the clauses of a program, a set of pairs of *contexts* $\Phi(l)$ that represents all occurrences of l in the proof-trees of the program in the following sense: for any occurrence of l in a proof-tree, there is a couple $(s_1,s_2) \in \Phi(l)$ such that s_1 represents the instantiation of l when it is created and s_2 represents its instantiation when it is satisfied. For an instantiation σ of l the context s of l representing σ contains two types of information: (i) which variables of l are ground by σ, and (ii) which variables of l are instantiated by σ to values sharing some variable. Unfortunately it is not possible, in general, to compute the pairs of contexts that represent precisely all occurrences of a literal. However one can construct an approximation of this set. Approximation that, hopefully, contains enough information for being useful in optimizing the execution of the program.

1 LITP Paris VII

2 Université de Bordeaux I, L.A.226 CNRS

3 Dip. di Matematica, Univ. di Padova e L.A.226 CNRS.

As already mentioned, the information that we propose to compute for any given program, is very close to that computed in [Son86], but, in addition to the difference in the domain of application, our approach differs from that of [Son86] for one important aspect. In [Son86] contexts are used in a static way, i.e., the tests of occur-check inserted in a clause do not depend on the particular occurrence of the clause. In contrast to this, we use our contexts dynamically, in the sense that, whenever a literal l is called, associated to l one finds its context that gives some information about the current instantiation of l, viz., (i) and (ii) above; information that would be too expensive to collect directly from the current instantiation and that can be useful in several ways, as explained below . Clearly, different occurrences of l will have different contexts taken from a finite set computed during the static analysis. This technique of dynamically using information computed statically is a classical one in the design of evaluators for attribute grammars [Rii83,Fil86]. In attribute grammars one computes statically several orders of the attributes of each nonterminal X and the evaluator dynamically chooses one order for each occurence of X.

Due to the restriction on the length of the paper, we omit the description of the static analysis (lengthy and technical) and concentrate on the optimizations that can be obtained by using the information computed by such a static analysis. This is possible since the interested reader may find in [Son86] a program analysis very close to that needed here. The static analysis that we propose and its differences and improvements w.r.t. that of [Son86] will be the topic of a future publication.

The optimization techniques studied are the AND-parallel execution of logic programs [Cha85a, DeG84], the intelligent backtracking [Bru84,Cod86a,Cox84,Mat85,Pie82] and the optimization of memory management in the sense of [War77]. For a simple AND-parallel execution of a logic program it is necessary to know, for each call of a clause, which of its literals share some variables. Contexts contain exactly this type of information. A similar information is necessary for doing intelligent backtracking, i.e., for not redoing deductions which are independent of the literal where the failure occurred. Finally, contexts can be used for detecting, for each call of a clause c, the variables of c that can be put into a local stack where they are destroyed as soon as no more choice is left below c. In [War77] this technique is applied for optimizing a Prolog compiler.

It is important to understand that the static analysis that we put forward can be done both before the interpretation and during the compilation of a logic program. The information produced can be used for optimizing the interpretation as well as for generating efficient code.

The rest of the paper is organized as follow. In Section 1 we recall some usual concepts. Section 2 contains the new definitions. Section 3 describes how contexts can be used for the 3 optimization methods cited above. Some concluding remarks close the paper.

1 Preliminaries

The reader is assumed to be familiar with the classical concepts of logic programming; for an introduction to the subject see for instance [Apt82,Llo84].

A <u>logic program</u> P (or just program) is a sequence of definite clauses [Apt82] of the form $A_0 \leftarrow A_1,\ldots,A_\alpha$ ($\alpha \geq 0$), A_0 is called the <u>head</u> of the clause and $A_1,\ldots,A_\alpha$ the <u>body</u> of the clause.
A <u>goal</u> is a clause of the form $\leftarrow B_1,\ldots,B_\alpha$, $\alpha \geq 1$. Let c be a definite clause or a goal, then for $j \in [1,\alpha]$, $\langle c,j\rangle$, denotes the j-th literal of its body (b-literal for short). If c is a clause then $\langle c,0\rangle$ is its head. LIT(P,G) is the set of all b-literals of the clauses of P and of the goal G. If t is a term, an atom, a clause or a goal, Var(t) is the set of variables appearing in t. As usual a <u>substitution</u> σ is a finite set of pairs:

$$\sigma = \{(x_1,t_1),\ldots,(x_n,t_n)\}$$

where all pairs have different first components and for no $j \in [1,n]$ $Var(t_j)$ contains x_i for some $i \in [1,n]$.
If V is a set of variables and σ a substitution then $\sigma/V=\{(x_i,t_i)$ s.t. $x_i \in V\}$

In this article with execution of a goal G in a program P we mean its execution in the Prolog sense, i.e., the selection rule used is that of expanding each time the left-most literal of the current goal and the clauses are tried in the order they have in P. This process can be viewed as a depth-first search of an SLD-tree, see [Apt 82]. In what follows we will call this tree the <u>SLD-tree of P and G</u>. The substitutions corresponding to the refutations of G found by this search process are called the <u>answer substitutions of P and G</u> .
Consider any path $n_0,\ldots,n_k$ in the SLD-tree of P and G, where n_0 is the root: each node of this path is labeled by a resolvent. We like to view such a path as a sequence $(t_0,\sigma_0),\ldots,(t_k,\sigma_k)$, where each t_i is a <u>proof-tree of P and G</u> [Cla79] and σ_i is a substitution that is said to be <u>associated to t_i</u>. Each couple (t_i,σ_i) is as follows.

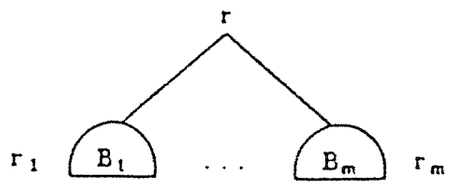

fig. 1

If the goal G is $\langle -B_1,..B_m$ then t_0 is the tree shown in fig. 1 and σ_0 is the empty substitution. Nodes $r_1,..,r_m$ of t_0 (r_i is the i-th son of node r) are called <u>open</u> because they must still be expanded. The couple (t_i,σ_i) is obtained from (t_{i-1},σ_{i-1}) by choosing an open node n of t_{i-1}, labeled by, say, $\langle c,j\rangle$, and expanding it; more precisely, one chooses a clause c' of P and unifies its head with $\sigma_{i-1}(\langle c,j\rangle)$, t_i is obtained from t_{i-1} as shown in fig. 2, and σ_i is obtained composing the m.g.u. of the expansion step to σ_{i-1}.

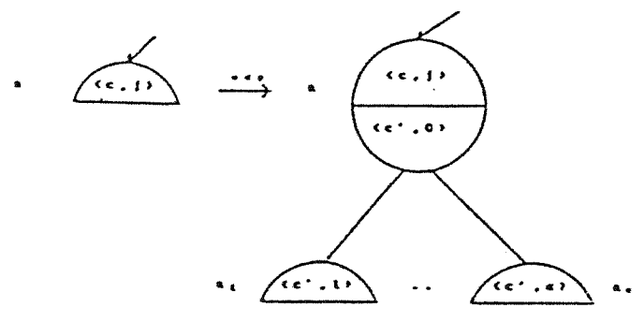

fig. 2

A node of a proof-tree with two literals, see n after expansion in fig. 2, is called <u>closed</u>. A closed node is <u>satisfied</u> if it does not have open nodes as descendants. In an expansion as that of fig. 2 one says that the nodes $n_1,...,n_\alpha$ are <u>created</u>. A proof-tree without open nodes is said to be <u>complete</u>.

2 New Definitions

Definition 1. Let P be a program and G a goal. For $\langle c,j\rangle \in LIT(P,G)$ a <u>context of $\langle c,j\rangle$</u> (and of the clause c or of the goal if c = G) is a triple $s = (W,B,E)$, where W and B are disjoints sets such that $W \cup B = Var(c)$ and $E \subseteq Var(c)^2$. The set of contexts of all b-literals of P and G is CON(P,G) □

Intuitively, a context of $\langle c,j\rangle$ represents a situation of an instance of c in a proof-tree t: the variables in B are ground (these are called <u>black</u>), those in W are not ground (they are called <u>white</u>) and $(X,Y) \in E$ means that at some moment of the construction of t the values of X and Y have shared at least one variable. Observe that if $(X,Y) \in E$ and, for instance, X is black that means that in the substitution associated to t, X and Y do not share any variable anymore, but that they did in the substitution corresponding to some previous expansion step.

A context of $\langle c,j\rangle$ is said to be on Var(c). For any 2 contexts $s_1 = (W_1,B_1,E_1)$ and $s_2 = (W_2,B_2,E_2)$ of two b-literals of one clause (or of the goal) we say that s_1 <u>contains</u> s_2, $s_1 \supseteq s_2$, if $E_1 \supseteq E_2$ and $B_1 \supseteq B_2$, we also say that s_1 <u>approximates</u> s_2, $s_1 \succ s_2$, if $E_1 \supseteq E_2$ and $B_1 \subseteq B_2$. Moreover, $s_1 \cup s_2$ is the context (W,B,E), where $B = B_1 \cup B_2$ $W = Var(c)-B$ and $E = E_1 \cup E_2$, and $s_1 \vee s_2$ is the context (W',B',E') where $B' = B_1 \cap B_2$, $W' = Var(c)-B'$ and $E' = E_1 \cup E_2$.
We will often regard contexts as graphs, thus talking about white and black nodes and of paths between 2 nodes. Contexts are very close to the A-substitutions of [Son86].
For any substitution σ, let V be the set of first components of the couples of σ. The context corresponding to σ is $\eta(\sigma) = (W,B,E)$, where $x \in B$ iff $\sigma(x)$ is ground, $W = V-B$ and $(x,y) \in E$ iff $\sigma(x)$ and $\sigma(y)$ share some variable.

Definition 2. For a given program P and goal G,
(i) a <u>context-relation</u> is a relation $\Phi \subseteq LIT(P,G)\times CON(P,G)^2$,
(ii) <u>up</u> and <u>down</u> <u>context-projections</u> are functions of type $LIT(P,G) \times CON(P,G) \times P \to CON(P,G)$, usually up and down context-projections are denoted by $\uparrow\Pi$ and $\downarrow\Pi$, respectively. □

Definition 3. Let P be a program and G a goal, a <u>context-description</u> for P and G is a triple $D = (\Phi,\downarrow\Pi,\uparrow\Pi)$, where Φ is a context-relation and $\downarrow\Pi$ and $\uparrow\Pi$ are down and up context-projections for P and G such that they satisfy the following condition:
consider any $\langle c,j\rangle \in LIT(P,G)$ and let $\langle c',0\rangle$ be unifiable with $\langle c,j\rangle$ where $c' = A_0 \leftarrow A_1...A_\alpha \in P$, let also $(\langle c,j\rangle,s,s') \in \Phi$ and $s'' = \downarrow\Pi(\langle c,j\rangle,s,c')$ then it must be that there exists at least one sequence of pairs of contexts $(s_1,s_1'),...,(s_\alpha,s_\alpha')$ such that $s_1 = s''$, for each $i \in [1,\alpha]$ $(\langle c',i\rangle,s_i,s_i') \in \Phi$, for $i \geq 2$ $s_i = s_{i-1} \cup s_{i-1}'$, and such that $\uparrow\Pi(\langle c,j\rangle,s_\alpha \cup s_\alpha',c') = s'$.
A sequence of pairs of contexts as above is said to be <u>relative to s_1 and c'</u>. □

A context-description for P and G can be used to associate contexts to the nodes of any proof-tree of P and G as explained below.

In a complete proof-tree for every internal node n two contexts are given: an input and an output one, denoted IN(n) and OUT(n) respectively. In an incomplete proof-tree all ancestors of the left-most open node have only the input context and all nodes at the left of these nodes have both input and output contexts, while the remaining nodes have nothing. The method is described inductively as follows:

(*i*) At the beginning the proof-tree for a given goal $G = \langle- B_1,\ldots,B_\kappa$ is as in fig. 1. $IN(r_1)$ is defined to be λ_G, where λ_G is the empty context on Var(G), i.e., (Var(G),∅,∅).

(*ii*) Assume that the proof-tree is expanded by unifying the (left-most) open node n labeled ⟨c,j⟩ with the head ⟨c',0⟩. We need to distinguish two cases:
1) c' has empty body
2) c' has at least one literal in its body.

1) One can compute $OUT(n) = \uparrow\Pi(\langle c,j\rangle,\lambda_{c'},c')$ and can propagate it to the satisfied ancestors of n : let $n_1,\ldots,n_k$ be these nodes, where n_1 is the father of n, n_2 is the father of n_1 and so on. n_1 is labeled by the couple $(\langle c'',f\rangle,\langle c,0\rangle)$ for some f,

$$OUT(n_1) = \uparrow\Pi(\langle c'',f\rangle,IN(n) \cup OUT(n),c)$$

For each $i \in [2,k]$, $OUT(n_i)$ is computed in a similar way. If n_k is the right-most son of r, i.e. B_κ, then the proof-tree is complete and we are done, otherwise one has to compute the input context of the node n' at the immediate right of n_k:

$$IN(n') = IN(n_k) \cup OUT(n_k)$$

2) Let n_1 be the left-most son of n:

$$IN(n_1) = \downarrow\Pi(\langle c,j\rangle,IN(n),c') \qquad \square$$

Definition 4. For a program P and a goal G, a context-description $D = (\Phi,\downarrow\Pi,\uparrow\Pi)$ for P and G is <u>complete</u> if the following condition holds: for any complete proof-tree t of P and G consider any node n, let ⟨c,j⟩ and ⟨c',0⟩ be the labels of n and let σ and σ' be the substitutions that were current when n was expanded and satisfied, respectively, it must be that $IN(n) \succ \eta(\sigma/Var(c))$ and $IN(n) \cup OUT(n) \succ \eta(\sigma'/Var(c))$ □

Note that it is easy, to find a complete context-description $D = (\Phi,\downarrow\Pi,\uparrow\Pi)$ for a program P and a goal G: it suffices that Φ contains for each b-literal ⟨c,j⟩ the triple $(\langle c,j\rangle,s_c,s_c)$, where $s_c = (Var(c),\emptyset,Var^2(c))$, and that for each clause c' unifiable to ⟨c,j⟩ $\downarrow\Pi(\langle c,j\rangle,s_c,c') = s_{c'}$ and $\uparrow\Pi(\langle c,j\rangle,s_{c'},c') = s_c$. Clearly such a description D would be of no use for optimizing the execution of G in P because it is a too pessimistic picture of the real situation. It is

not difficult to device an algorithm that computes more realistic context-descriptions. Such an algorithm can be found in [Son86].

3 Optimizations

The idea of using a context-description $D = (\Phi,\downarrow\Pi,\uparrow\Pi)$ for optimizing the execution of programs is simple: when expanding a node n labeled by ⟨c,j⟩ of a proof-tree by means of a clause c' $A_0 \langle- A_1,\ldots,A_\kappa$, from IN(n) and $\downarrow\Pi$ we know $IN(n_1)$ and then, from the context-relation Φ one can construct all sequences of pairs of context $(s_1,s_1'),\ldots,(s_\kappa,s_\kappa')$ relative to s_1 and c', cf. Def 3. In this way one has an overview of all the possible relations among the variables of c' that may exist in the deductions below n. Such information is the base of many optimization techniques.
In what follows we consider three of them : AND-parallel execution of programs [Con85,DeG84], intelligent backtracking [Cha85b,Pie82,Cod86a] and finally a space saving technique.

A) AND-parallelism

The execution of logic programs can be improved if one examines in parallel alternative solutions (OR-parallelism) and/or the subparts of one solution (AND-parallelism). In the case of AND-parallelism, before expanding in parallel literals of one clause one would like to be sure that they do not share any variable, otherwise a variable binding conflict may arise if the different processes instantiate the same variable to different values. Two literals of an instance of a clause that satisfy this property are called independent. Clearly the simple analysis of one clause is not sufficent to know whether, when the clause is used, any 2 b-literals will be independent. Example 1 explains this phenomenon.

Example 1 Consider the clause c: p(X,Y) <- p(X),q(Y). The two b-literals of c have no variable in common but if c is used in a proof tree with the current substitution σ such that σ(X) = f(Z) and σ(Y) = g(Z), it is clear that the two b-literals of c cannot be solved in parallel (without facing possible conflicts) □

At this point it should be already evident that contexts are exactly the information one needs for solving the type of problem shown in Ex. 1. Let us see how this is done. Let P and G be a program and a goal and $D = (\Phi,\downarrow\Pi,\uparrow\Pi)$ a complete context-description for them. Consider the proof-tree t whose left-most open node n is labeled by

$\langle c,j\rangle$ and assume that the context $IN(n)$ is s. Assume also that the expansion of n using the clause c' : $A_0 \leftarrow A_1 \ldots A_\alpha$ succeeds and that $\downarrow\pi(\langle c,j\rangle,s,c') = s_1$. One wants to know the subsets of b-literals of c' that can be executed in parallel in all computations that may solve $\langle c,j\rangle$.

Due to the fact that the static analysis is based on the standard PROLOG selection rule (i.e., the left-most literal is chosen), see [Son86], with the context-description computed by this analisys one can "parallelize" a clause as c' only in the sense that its body can be divided in groups, $A_1,\ldots,A_{m(1)}; A_{m(1)+1},\ldots,A_{m(2)};\ldots; A_{m(k)+1},\ldots,A_\alpha$; such that the literals of each group can be executed in parallel, but the groups must be executed sequentially in the order that they have in c'.

We give two methods for doing a correct "slicing" of c' : a simple one called the <u>backward method</u> and a smarter one called the <u>forward method</u>. Let us first define the following contexts :
Let $(s_1,s_1^{\cdot j}),\ldots,(s_\alpha^j,s_\alpha^{\cdot j})$, $j\in[1,m]$, be all sequences of pairs of contexts relative to s_1 and c'. For $k\in[1,\alpha]$ the <u>k-th context s_k for s_1 and c'</u> is as follows : $s_k = \nabla\{s_k^j \,/\, j\in[1,m]\}$. The <u>final context for s_1 and c'</u> is $s_{fin} = \nabla\{s_\alpha^j \cup s_\alpha^{\cdot j} \,/\, j\in[1,m]\}$.
Two literals A_k and A_j are <u>forward independent w.r.t. a context s</u>, if in s there is no path (possibly empty) between two white variables X and Y of A_k and A_j respectively.
A_k and A_j are <u>backward independent w.r.t. s</u> if in s there is no path between (any) two variables X and Y of A_k and A_j, respectively.

The first "slicing" method is as follows : the body of c' is divided into groups $A_1,\ldots,A_{m(1)};\ldots;A_{m(k)+1},\ldots,A_\alpha$; such that the literals of each group are pairwise backward independent w.r.t. s_{fin} (the final context for s_1 and c').
Clearly, this method is very pessimistic because the slicing is done using the last context s_{fin}. Therefore one may expect it to be not very interesting in practice.

An improved method is the following : the body of c' is sliced into groups as above where the first group is the maximal prefix of the body of c' such that all its literals are pairwise forward independent w.r.t. s_1, those of the second group are all pairwise forward independent w.r.t. the $(m(1)+1)$-th context for s_1 and c' and so on for all other groups.

Example 2. Let P and G be a program and a goal and D be a complete context-description for them. Consider an occurrence of a cluse c': $A_0(X,Y,Z) \leftarrow A_1(X)\ A_2(Y,Z)\ A_3(X,Y)\ A_4(X,Z)$, in a proof-tree, and let n_i be the node labeled by A_i, $i\in[0,4]$. Assume that

$IN(n_1) = s_1 = (\{X,Y,Z\},\emptyset,\{(Y,Z)\})$ and let D be such that the contexts s_2, s_3, s_4, and s_{fin} for s_1 and c' are as follows:
$s_2 = (\{Y,Z\},\{X\},\{(Y,Z)\})$, $s_3 = s_4 = (\{Z\},\{X,Y\},\{(Y,Z)\})$ and $s_{fin} = (\emptyset,\{X,Y,Z\},\{(Y,Z)\})$.
The backward method slices the body of c' in 3 groups: $A_1 // A_2 \& A_3 \& A_4$.
The forward method slices it in only 2 groups : $A_1 // A_2 \& A_3 // A_4$. □

Assume, for simplicity, that c' has been sliced (using either method) into only two groups : $A_1,\ldots,A_m$ and $A_{m+1},\ldots,A_\alpha$. For the methods to work at the execution of each literal one must know its input context. It is not difficult to see that from s_1 one can find a convenient input context for each $A_1,\ldots,A_m$. One can also see that, when the execution of $A_1,\ldots,A_m$ is completed, merging s_1 with the output contexts of $A_1,\ldots,A_m$, produces the input context needed for the execution of $A_{m+1},\ldots,A_\alpha$.
We stress the fact that for both the methods described above the slicing of a clause c' is done statically because it depends only on c' and on the input context s_1, taken from a finite set.

B) Intelligent Backtracking

Consider a program P and a goal G. As mentioned in sect.2, the execution of G in P consists in traversing the SLD-tree of P and G in a depth-first fashion. This is implemented as follows : when a node, say n, is found that cannot be expanded (all clauses have already been tried) one backtracks to the first ancestor n' of n that still has some alternatives to be tried, i.e., inexplored paths in the SLD-tree.
In Prolog one applies this backtracking method systematically, even when the remaining alternatives of n' have no chance of modifying the causes of the failure of n. Several methods have been proposed for improving this naive backtracking, most of these techniques, [Bru84,Cod86a&b,Cox84,Mat85,Pie82], are dynamic in the sense that all the work is done during the execution/interpretation of a program.
Roughly, these methods consist in connecting each "piece" of the computed substitution with the expansion that has caused it. It is not sure that these techniques really bring an improvement: the extra computation they require may be more expensive than the useless expansions that they allow to skip, see also [Bru84,Cha85b] for some statistics. In contrast to the heaviness of the dynamic methods [Cha85b] propose a much simpler static technique that is described in example 3.

Example 3 Consider a program P and a goal G: <- p(X),q(Y),w(X) and assume that in a proof-tree the son r_3 of the root, labeled by w(X), fails: one has to backtrack to the first node (with some unexplored alternatives) to the left of r_3 that has a chance of modifying the current value of the variable X. It is easy to see that no descendent of r_2 satisfies this condition, q(Y) may modify the value of Y but not that of X! Hence one can safely backtrack to the right-most descendent of r_1 that has some untried alternatives.
Assume now to have the clause p(X,Y) <- q(X),w(Y). If (the instance of) w(Y) fails due to one of its descendents, one may want to backtrack directly to the father, but this may imply that some existing answer substitutions are never found if the current values of X and Y have some variable in common. Clearly, such an information is contained in a complete context-description of the program.□

The need of context-descriptions is mentioned also in [Cha85b] eventhough no detail is given there about the way they could be constructed and used.

Let us see the method that we propose: Let P be a program and G the goal and consider an instance of a clause c: A_0 <- $A_1,\ldots,A_\alpha$ of P in a proof-tree of G in P, where $n_0,\ldots,n_\alpha$ are the nodes containing $\langle c,0\rangle,\ldots,\langle c,\alpha\rangle$, respectively. Assume that n_i fails for $i \in [1,\alpha]$.
A <u>backtracking node</u> (b-node for short) is any n_k, $k \in [1,i-1]$ such that either there is a variable X of n_i such that X is white in $IN(n_k)$ and black in $OUT(n_k)$, or in $IN(n_i)$ there is a path between a white variable X of n_i and Y of n_k (including the case X = Y). One must backtrack to the right-most descendent (with alternatives) of the right-most b-node of n_i. The remaining b-nodes, if any, must be kept in a set, called $B(n_0)$, associated to n_0. $B(n_0)$ is useful in case some other node n_f, $f \in [1,\alpha]$ fails, i.e., it is not the first time that a node in $\{n_1,\ldots,n_\alpha\}$ fails.
Thus, in general, if n_i fails one computes its b-nodes using $IN(n_1),\ldots, IN(n_i)$, chooses the right-most node n_k, $k \in [1,i-1]$ among them and the nodes in $B(n_0)$ and updates $B(n_0)$ adding to it the remaining b-nodes of n_i. $B(n_0)$ is set to $\emptyset$ when $n_1,\ldots,n_\alpha$ are created and it is set again to $\emptyset$ when n_0 is satisfied.

One needs to store the remaining b-nodes in $B(n_0)$ in order to guarantee the completeness of the method w.r.t. the naive backtracking. This need is already present in the dynamic method of [Bru84,Cod86b], whereas in [Cha85b] it is avoided, at the expence of the accuracy of the backtracking, by statically computing the most pessimistic b-nodes.

Example 4. Consider, as usual, a program P, a goal G, and a complete context-description D for P and G. Assume that the clause c': $A_0(X,W)$ <- $A_1(W)\ A_2(Y)\ A_3(Y,Z)\ A_4(X,Y)$, of P occurs in a proof-tree t and that each predicate A_i labels node n_i of t, $i \in [0,4]$. Let n_4 be the left-most open node of t and assume that it has no more alternatives to try, i.e., A_4 fails. Assume also that $IN(n_i)$, $i \in [1,4]$, are as follows: $IN(n_1) = (\{X,Y,W,Z\},\emptyset,\{(X,W)\})$, $IN(n_2) = (\{X,Y,Z\},\{W\},\{(X,W)\})$, $IN(n_3) = (\{X,Z\},\{Y,W\},\{(X,W)\})$, and $IN(n_4) = (\{X\},\{Y,W,Z\},\{(X,W)\})$.
Node n_4 has two b-nodes : n_1 and n_2. Node n_3 is not a b-node because the variable Y, that A_3 and A_4 share, is already black in $IN(n_3)$. □

The method just presented has a limitation: when a b-node n is chosen one backtracks to the right-most descendent n' of n that has some untried alternatives. One would like to choose the descendent of n to which to backtrack in a more intelligent way, in the sense, that it is not sure that changing the subtree rooted in n' will change the values of the variables of the literals in n.

Extending our method in this way seems not too difficult. Roughly, it can be done as follows: one may modify the unification algorithm in such a way that, in case of failure in unifying two literals $\langle c,j\rangle$ and $\langle c',0\rangle$, it specifies the variables of c and c' the actual values of which have caused the failure. In addition to this one needs a recursive procedure on proof-trees that works as follows. Consider a node n labeled by $\langle c,j\rangle$ and $\langle c',0\rangle$, where c' has the form A_0 <- $A_1,\ldots,A_\alpha$. Assume that the procedure arrives in n with a subset V of Var(c). One wants to find the right-most son n_i such that, if one changes the subtree rooted in it, the values of some variables in V may change.

The procedure must construct the set V' of variables of c' that are dependent on some variables of V. Two variables X and Y of c' and c, resp., are <u>dependent</u> if the following holds: let σ be the m.g.u. of $\langle c,j\rangle$ and $\langle c',0\rangle$, $\exists$ a term t such that either it contains Y and is such that $\sigma(X) = \sigma(t)$, or it contains X and is such that $\sigma(Y) = \sigma(t)$. In the case that the second possibility only holds, we say that X is <u>useful</u> for Y.

Example 5. Consider the unification of the b-literal $A_0(X,f(Z,Z),Z,b)$ with the head of the clause c': $A_0(f(W),Y,a,Y')$ <- $A_1(W,g(Y),W')\ A_2(W,W',g(Y'))$. Variable W is useful for X, and Y and Z are dependent.□

Let us go back to our recursive procedure. Once V' is defined, the node n_i is clearly the right-most node in $\{n_1,\ldots,n_\alpha\}$ such that in

$IN(n_i)$ there is a path (possibly empty) between 2 whites variables X and Y s.t. $X \in A_i$ and $Y \in V'$. Let V_i be the set of variables of A_i as X above. One must recursively call the procedure on n_i with parameter V_i.
The recursive call stops when a terminal node is found or when it is in a node n such that no son of n can be selected according to the above rules. The node at which it stops is where one has to backtrack.

A similar recursive procedure is described in the backtracking method of [Dem85]. The information that directs the recursion in the method of [Dem85] is also computed statically, but, in contrast with our context-descriptions, consists in properties that the program must satisfy and, thus, the method can be applied to a restricted family of logic programs, that introduced in [Der85]. In [Dem85], intelligent backtracking is done together with AND-parallel execution. It is easy to see that the backtracking technique presented above is compatible with the methods of guiding the AND-parallelism described in [A] above : any b-node n_k of a node n_i is not independent of n_i and hence n_i and n_k are executed sequentially.

C) Space-saving techniques

The distinction between global and local variables of a clause has been made in [War77] with the goal of optimizing the space efficiency of the execution of compiled Prolog programs. Consider the instance of a clause c in the schematic proof-tree shown in fig.3.

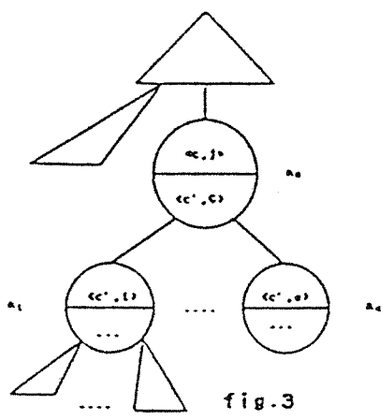

fig.3

Suppose that n_0 is satisfied and that in the subtrees rooted in $n_1,\dots,n_k$ there are no unexplored alternatives. The variables of c' that are not needed for instantiating any variable of c can be thrown away, because they will not be referenced anymore. We call them the <u>unreferenced</u> variables of this occurrence of c' . It would be convenient to keep these variables in a (local) stack different from

that of the referenced variables, in order to recover the space they occupy as soon as the above described situation of c' occurs.

A subset of the unreferenced variables of all occurrences of a clause c' are the <u>local</u> variables of c' as defined in [War77]: local are the variables that have multiple occurrences in c' with at least one in the body and none in a compound term.

A complete context-description allows to distinguish another subset of the unreferenced variables of any occurrence of a clause c' in a proof-tree : the subset of its <u>unimportant</u> variables. Clearly, different occurrences of c' may have different sets of unimportant variables.
Consider the proof-tree obtained from that of fig.3 by making $n_1,\dots,n_k$ open again. The subset of the unimportant variables of c' is defined as follows. Let V be the set of variables of c' that are useful (see the end of point (B)) for some variable of c. V' is obtained eliminating from V each variable X such that every variable of c, for which X is useful, is black in $IN(n_0)$. Finally, consider the final context s_{fin} for $IN(n_1)$ and c'; a variable X of c' is unimportant (for the considered occurrence of c') if there is no path (possibly empty) in s_{fin} between X and a variable of V'.

Example 6. Recall example 5 and assume that the literal <c,j> occurs in a proof-tree and that it is expanded using clause c'. As usual, the nodes of this occurrence are n_0, n_1, and n_2. Only variable W of c' is useful for a variable of c, viz., for X. Hence, if in $IN(n_0)$ X is black, all variables of c' are unimportant, whereas, if X is not black, then one must look at the final context s_{fin} for $IN(n_1)$ and c'. Assume that $s_{fin} = (\{W'\},\{Y,Y',W\},\{(W,W')\})$. In this case only Y and Y' are unimportant for the occurrence of c'. □

Observe that in general local and unimportant variables are incomparable sets. In practice, one can do the following: when a node n_0 is expanded by means of a clause c', one can put in a special stack the local and the unimportant variables (this last set depends on $IN(n_1)$ and on c' only). Obviously following [War77] void and temporary variables will not ever appear on this local stack and will be thrown away after unification.

Conclusion We have described a general method of using statically collected information for optimizing the execution of logic programs. The most important features of this method are that it uses the static information in a dynamic way and that the same information is the basis of all the considered optimization techniques.

In the future we intend to formally define an algorithm for computing complete context-descriptions closer to the reality than those of the algorithm of [Son86]. The goal being that of further exploiting the optimization techniques presented here (and eventually new ones). It is in our plans to implement this algorithm in order to test it against real life programs.

References

[Apt 82] K.R. Apt, M.H. Van Emden. Contributions to the theory of logic programming; JACM vol.29 (1982),841-862.

[Bru 84] M. Bruynooghe, L.M. Pereira. Deduction revision by intelligent backtracking; Implementations of Prolog, Campbell (ed.), Ellis Horwood (1984), 194-215.

[Cha 85a] JH. Chang, A.M. Despain, D. DeGroot. AND-Parallelism of logic programs based on a static data dependency analysis; Digest of papers of COMPCON Spring' 85, (1985), 218-225.

[Cha 85b] JH. Chang, A.M. Despain. Semi-intelligent backtracking of Prolog based on a static data dependency analysis; Int. Symp. on Logic Programming (1985), 10-21.

[Cla 79] K.L. Clark. Predicate logic as a computational formalism; Research report, Dept. of Computing, Imperial College, London (1979).

[Cod 86a] C. Codognet, P. Codognet, G. Filé. A very intelligent backtracking method for logic programs; ESOP, Saarbrück, LNCS 213 (1986), 315-326.

[Cod 86b] C. Codognet, P. Codognet, G. Filé. Backtracking intelligent en programmation logique; Symposium sur la programmation en logique, CNET, Tregastel (1986).

[Con 85] J.S. Connery, D.F. Kibler. AND-Parallelism and nondeterminism in logic programs; New generation computing, vol. 3 (1985), 43-70.

[Cox 84] P.T. Cox. Finding intelligent backtrack points for intelligent backtracking; Implementations of Prolog, op. cit., 216-233.

[DeG 84] D. DeGroot. Restricted AND-Parallelism; Proc. of the Int. Conf. o (1984), 471-478.

[Dem 85] P. Dembinski, J. Maluszynski. AND-Parallelism with intelligent backtracking for annoted logic programs; Int. Symp. on Logic Programming (1985), 29-38.

[Der 85] P. Deransart, J. Maluszynski. Relating logic programs and attribute grammars; J. Logic Programming, vol. 2 (1985), 119-155.

[Fil 86] G. Filé. Classical and incremental evaluators for attribute grammars; Proc. 11-th CAAP, LNCS 214 (1986), 112-126.

[Llo 84] J. W. Lloyd. Foundations of logic programming. Springer Verlag, series in Symbolic Computation, (1984).

[Mat 85] S. Matwin, T. Pietrzykowski. Intelligent backtracking in plan-based deduction; IEEE PAMI vol.7 no.6, (Nov 1985)

[Mel 86] C. S. Mellish. Abstract interpretation of Prolog programs; Int. Conf. on Logic Programming, LNCS 225 (1986), 463-474.

[Pie 82] T. Pietrzykowski, S. Matwin. Exponential improvement of exhaustive backtracking, a strategy for plan-based deduction; 6-th CADE, LNCS 138 (1982), 223-239.

[Rii 83] H. Riis Nielson. Computation sequences: a way to characterize subclasses of attribute grammars; Acta Informatica 13 (1983), 255-268.

[Son 86] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction; ESOP, Saarbrück, LNCS 213, (1986), 327-338.

[War 77] D. Warren. Implementing Prolog - Compiling predicate logic programs; Research reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, (1977).