ANALYSIS OF DEPENDENCIES TO IMPROVE THE BEHAVIOUR OF LOGIC PROGRAMS.

Maurice Bruynooghe

Afdeling Toegepaste Wiskunde en Programmatie

Katholieke Universiteit Leuven

B-3030 Heverlee/Belgium

Abstract

Traditionally, backtracking uses a total order over the derivation steps. On failure, it returns to the most recent state. We consider states as a set of derivation steps. For each step, we save a 'inputset' validating the derivation. This results in a partial order over the derivation steps and allows a more accurate 'intelligent' backtracking.

1. Introduction

The goal statement of a logic program comprises a set of problems [procedure calls] which have to be solved simultaneously. For each problem, there exists a set of alternative rules [procedure definitions]. The successful application of a rule to a problem reduces the problem to a [possibly empty] set of subproblems. It is common practice to use a depth first search combined with backtracking to explore the search space [and- or tree]. However, backtracking ignores the dependencies between different subproblems and very systematically explores the complete search space. This results in a sometimes stupid behaviour when backtracking reacts on the failure to solve a subproblem with an attempt to find another solution for a subproblem which is completely irrelevant to the detected failure, i.e. the behaviour condemned by Sussman and McDermott [8] of a robot attempting to pick up an object with the left hand after a trial with the right hand failed because the object was too hot. In this paper, we show how an analysis of the dependencies between different subproblems can be used to obtain a substantially better search behaviour.

2. Logic programs [5], [6]

A logic program comprises a set of procedures and a goal statement. A goal statement has the form $\leftarrow A_1, \dots, A_n$ ($n \geq 1$) with the symbols A_i procedurecalls ['literals']. A procedure has the form $B \leftarrow A_1, \dots, A_n$ ($n \geq 0$). The heading B and the calls A_i of the body are again literals.

A procedure can be used to execute a call when the heading matches that call ['resolution' [7]]. To execute a goal statement $\leftarrow A_1, \dots, A_n$, a call A_i is selected, [in PROLOG [3], [9], the leftmost]. A procedure $B \leftarrow B_1, \dots, B_m$ matching that call with substitution θ is taken and a new goal statement $\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_m, A_{i+1}, \dots, A_n) \theta$ is derived. A solution is obtained when an empty goal statement is derived. The answer consists of the composition of the successive substitutions.

Backtracking is necessary because some calls A_i match different procedures.

3. Backtracking

The computation consists of a sequence of states or goal statements starting with the given goal statement. The successor of a state is derived by applying a procedure on the selected call. Once all solutions derivable from a given state are obtained, the backtracking algorithm returns to the previous

state and attempts to apply a yet untried procedure on the call selected in that state. This search process does not consider the dependencies between the different calls of the successive goal statements. It simply assumes that each step depends on the whole state.

To improve the search, we consider the state not as a whole but as a set of derivation steps. Each step being the result of applying a procedure on a call and resulting in a substitution and in the creation of instances of the calls in the body of the applied procedure. Having refined the concept of state, we can imagine that a new step of the computation only depends on a subset of the previous steps. Let us call this subset the inputset. The inputset of a derivation step - an attempt to apply a definition on a call - consists of all previous steps containing information (substitutions - created calls) which are

- either necessary to generate the substitutions resulting from the unification between the call and the heading.
- or necessary to conclude that the definition is inadequate for the given call.

In other words, the inputset contains all steps crucial to the validity of the computed results of the performed derivation step.

To exploit the information about the inputset, we have to save it, i.e. for each executed call, we not only need the set of available (yet untried) definitions and the substitution resulting from the applied definition but also the inputsets of the applied and eliminated definitions.

Assuming we can determine the inputsets, we can write an algorithm for 'intelligent' backtracking.

Algorithm

Repeat

- select a unexecuted call.
- if there are some definitions available
 - then
 - . select a definition and remove it from the set of available definitions
 - . unify the call with the heading of the selected definition and determine the inputset
 - . if the unification is successful
 - then . store the substitution and the inputset of the applied definition
 - . create instances of the calls in the body of the applied procedure and initialise their sets of available definitions
 - else ['shadow' backtracking]
 - . add the selected definition with its inputset to the set of eliminated definitions.
- else [exhaustion of the available definitions - 'deep' backtracking]
 - . the set of 'suspects' for this failure is the union of the inputsets of the eliminated definitions.
 - Undoing one of the steps in this set will at least return one of the eliminated definitions to the set of available definitions and will eventually cure the failure.
 - . to obtain a systematic generation of the potential solutions, it is necessary to select as 'culprit' the 'last' one in the set of suspects. As 'last' one we mean any of the suspects which is not in the inputsets (of the applied and eliminated definitions) of the other suspects.

the derivation performed by the 'culprit' is deleted (substitution - created calls), the applied definition is added to the set of eliminated definitions with its inputset extended by all other suspects, indeed the incompatibility of the substitution generated by this step with the other suspects caused the rejection of the definition.

All computations partly based on the computation performed by the culprit are no longer valid and must be deleted :

- .. The application of a definition on a call must be undone and the definition returns to the set of available definitions when the culprit belongs to the inputset (In turn, other steps based on this deleted steps need to be deleted)
- .. An eliminated definition returns to the set of available definitions when the culprit (or another deleted step) belongs to the inputset.

until a solution is found.

Note on the selection of calls

Usually, the same call will be selected after shadow backtracking while, after deep backtracking, the call associated with the culprit will be selected.

Differences with the naïve backtracking algorithm

The naïve algorithm simply returns to the previous call on deep backtracking. However, all attempts to find a solution are doomed to fail as long as all information contributing to the first failure remains in the system. It means that the naïve algorithm finally will return to a possible culprit.

The intelligent algorithm not only returns immediately to the culprit, it also saves the still valid information about some eliminated and applied definitions. The naïve algorithm deletes this information and, eventually, will rediscover the unsuitability of the eliminated procedures and has to recompute the substitutions of the applied definitions.

A partial order

The naïve algorithm defines a total order over the steps of the computation i.e. the order of their execution. The intelligent algorithm defines only a partial order i.e. a step must be executed after the steps in the inputsets of the applied and eliminated definitions. As a result, a particular state can be obtained by different orders of execution. It is even possible and meaningful to execute different steps in parallel (see further). Selection of the culprit has to be such that the partial order over the steps remains.

A simple example

We can already give a simple example, even before discussing the determination of inputsets. It is easy to convince oneself that the given inputsets are sufficient to perform the given steps and thus result in a correct behaviour of the algorithm.

```

<- FP(u,v,w)
a. FP(x,y,z) <-- Q(x), T(z), R(y,y), P(x,y)
b. Q(A) <--
c. Q(B) <--
d. R(A,A) <--
e. R(s,t) <-- SR(s,t)
f. SR(B,A) <--
g. P(B,A) <--

```

- h. $P(A,B) \leftarrow$
- i. $T(C) \leftarrow$
- j. $T(D) \leftarrow$

Applying definition [a] on the only call $FP(u,v,w)$ results in the substitution $\{x \leftarrow u, y \leftarrow v, z \leftarrow w\}$ and the creation of the subgoals $Q(x)$, $T(z)$, $R(y,y)$ and $P(x,y)$. Execution of $Q(x)$ with FP as inputset gives, with definition [b], the substitution $\{u \leftarrow A\}$. In a similar way we can use FP as inputset to apply definition [i] on $T(z)$ - which gives $w \leftarrow C$ - and to apply definition [d] on $R(y,y)$ - which gives $v \leftarrow A$.

Applying definition [g] on $P(x,y)$ with FP and Q as inputset results in a failure, indeed x does not unify with B because of the existing components $x \leftarrow u$ and $u \leftarrow A$. Applying definition [h] on the same call, but with FP and R as inputset also results in a failure. The set of available definitions for the call P being exhausted, we need deep backtracking. The set of suspects consists of FP , Q and R . Q and R can be selected as culprit. Suppose we select R ; its applied definition [d] becomes eliminated due to the other suspects i.e. FP and Q . Undoing the execution of R affects the computations based on R ; i.e. the elimination of definition [h] for the call $P(x,y)$. This definition becomes again available, however the other definition [g] remains eliminated.

Applying definition [e] on $R(y,y)$ with inputset FP results in the substitution $\{s \leftarrow v, t \leftarrow v\}$ and the subgoal $SR(s,t)$. An attempt to solve SR with R and FP in the inputset and using the only definition [f] fails. Deep backtracking with FP and R as suspects results in undoing the call R : the definition [e] is eliminated and the inputset consists of FP . R has exhausted its set of available definitions and we again have deep backtracking. The suspects are FP and Q . Q must be the culprit and definition [b] is eliminated by FP (inputset). Now definition [d] becomes again available for call R (but not e) and definition [g] for call P . Also the execution of call $T(z)$ remains intact although executed after Q . The state of the computation is summarized in Fig 1.

Fig 1. The computations takes a new start from the call $Q(x)$. A substantial amount of information about the calls T and R has been saved. This picture represents the and-tree of subproblems. The calls are labeled by numbers. $ap: i \{1\}$ means that definition [i] is applied with call [1] i.e. FP , as inputset. $el: b \{1\}$ means that definition [b] is eliminated and the inputset consists of call [1]. $av: \{g,h\}$ means that definitions [g] and [h] are available.

This new attempt starting with call Q will lead to a solution. The naïve algorithm will not only loose the information about the calls T and R , it will also make a completely irrelevant attempt starting with procedure [j] for call $T(z)$. Remark that $Q(x)$ and $T(z)$ both do not share variables with the failing call $R(y,y)$. It is the history of previous failures [$P(x,y)$] which makes $Q(x)$ a valid backtrackingpoint and $T(z)$ an irrelevant one. We conclude that any

serious attempt to apply 'intelligent' backtracking needs to consider the history of the successive failures.

4. Determination of the inputsets

Taking the inputsets to big does not result in an incorrect behaviour, it only makes the backtracking and thus the search for a solution less accurate.

As an extreme, we could take all previously executed calls as inputset. In this case, the algorithm will always select the most recent call as culprit (the other suspects are all in the inputset of the most recent one) and the algorithm reduces to the naïve backtracking algorithm.

At least, the inputset contains the father of the considered call and the inputset of the procedure applied to it (a.o. the other ancestors). Indeed, these elements are responsible for the mere existence of the call. It also means that the unification algorithm at least has access to the substitutions associated with these elements.

Which other information is needed by the unification algorithm is a question with different possible answers. We consider two seemingly interesting answers.

- a. When the unification algorithm has to unify a variable x with a term t , it will create the component $x \leftarrow t$ unless there exists a component $x \leftarrow t'$ in which case the algorithm has to unify t' with t . Thus, whenever the unification algorithm accesses the binding of a variable, it adds the call generating that binding and the inputset of the procedure applied on that call to the inputset of the call being executed. Some non-determinism is possible when unification fails. For example, the failure of the unification between $P(x,y)$ and $P(A,B)$ can be detected, either by accessing a binding $x \leftarrow B$ or by accessing a binding $y \leftarrow A$.
- b. When unifying x with t we can generate $x \leftarrow t$ even when a component $x \leftarrow t'$ exists but the generator does not belong to the inputset (to the ancestors). However in that case, we can have different substitutions for the same variable and we have to verify their compatibility. (Maybe by adding a special subgoal $\text{Equal}(t,t')$ having both generators as its initial inputset). This is a rather complex task - t and t' can contain variables which in turn can get different bindings - but this approach has some advantages :
 - As a result of unification, the inputset solely consists of the ancestors. This gives more independence between subgoals.
 - The behaviour is less dependent on the order of selection of the calls. Indeed most conflicts arise from inconsistencies between different substitutions. The set of suspects consists of the generators (and their inputsets) of these conflicting substitutions and - as far as the solution of earlier conflicts allows - any of these generators can be chosen as culprit. With method a, the last generator, the one completing the conflicting set, will read the already existing substitutions and the unification will fail. As a result, the last generator is the culprit (in fact shadow backtracking). Method b is more neutral to the order in which the generators are executed and can choose as culprit the one which gives the most promising (smallest) search space (especially interesting where the user is unaware of the desired control i.e. theorem proving).
 - This approach is attractive in the context of parallelism : some processes performing unification, only using the information of the ancestors; other processes checking (read only!) the consistency of different substitutions.

Optimising the manipulation of the inputsets

The construction of the inputsets is such that, whenever a call {b} is in the inputset of the procedure applied on a call {c}, also the elements in the inputset of the procedure applied on {b} belong to the inputset of the procedure applied on {c}. This is in fact a transitive relationship between the different steps of the execution. Without loss of information, we can delete any call {a} from the inputset of a call {c} when {a} also belongs to the inputset of a definition applied on another call belonging to the inputset of {c}. Indeed, computing the closure of the relation restores the original information. This not only reduces the amount of information to be stored but also simplifies the computation of the culprit. Indeed, only the explicitly listed elements can be last in the set of suspects and are possible culprits. It is not necessary to compute the whole set of suspects.

With both of the above methods, the inputset becomes initialised by the father of the call. With method a, it is extended with the generator of a substitution $x \leftarrow t$ whenever the unification algorithm needs that substitution and the generator is not yet in the implicit inputset.

When all definitions of a particular call are exhausted [deep failure), it is sufficient to take as suspects the explicitly known elements of the inputsets of the eliminated procedures [the set of the original algorithm can be derived by transitive closure). The inputset of the culprit 's rejected definition need only be extended with the other possible culprits [again the transitive closure gives all elements). The invalid information can be removed by initialising a set of deleted steps with the culprit. Any eliminated procedure with an element of the set of deleted steps on its inputset returns to the set of available procedures. Any step with an element of the set of deleted steps in the inputset of the applied procedure is undone [substitution removed, definition becomes available) and added to the set of deleted steps.

With method b, when an inconsistency is detected, the set of suspects consists of all generators of substitutions contributing to the inconsistency. The further treatment is similar to the case of deep failure.

Reduction of the number of calls

If the set of available definitions is empty and the explicit inputset of the eliminated and applied definitions consists of only one and the same element, then we can discard that call and its substitution can be added to the substitution of the only call in its inputsets. This will not substantially change the backtracking behaviour because selection of that call as culprit immediately results in another failure followed by deep backtracking and, in the new set of suspects, the call is only replaced by the single element of its inputsets. Of course, the discarded call is again initialised when the call absorbing the discarded call is undone by the backtracking.

A simplification

A simpler but less accurate version is obtained by giving the eliminated and applied procedures the same inputset, namely the union of their respective inputsets. For each call, only one inputset needs to be stored. This approach results in the method described in [1].

5. Example

As an example, we take the eight queens problem, however, to simplify the exposition, we restrict ourself to three queens and a square board of side

three. The observation that each row and each column need to have exactly one queen allows us to write a configuration as a list of numbers where the i th number gives the columnnumber of the queen on the i th row. In our case a possible solution is a permutation of the numbers 1 to 3. To be a solution, two queens have to be on different diagonals. Solutions can be computed by the goal statement

```
<- Perm [1.2.3. Nil, q), Safe [q)
[1.2.3. Nil stands for the term .[1, .[2, . [3, Nil)))) where Perm expresses
the binary permutation relation and safe the unary relation describing all configurations with the queens on different diagonals.
```

Using a ternary relation $\text{Del}(e, l_1, l_2)$ which is true when the list l_2 is derived from the list l_1 by removal of the element e , we can define the permutation as follows.

```
a. Perm [Nil, Nil) <-
b. Perm [x.y, u.v) <- Del[u, x.y, w), Perm[w, v)
c. Del [x, x.y, y) <-
d. Del [u, x.y, x.v) <- Del[u, y, v)
```

Using a ternary relation $\text{Check}[p, l, d)$ to compare a queen p with the queens of list l with d the distance between the row of p and the row of the first queen on l and another ternary relation $\text{Nodiag}[p, q, d)$ to compare p with queen q and d the distance between the rows of p and q , we can write Safe as :

```
e. Safe [Nil) <-
f. Safe [p.q) <- Check [p, q, 1), Safe [q)
g. Check [p, Nil, n) <-
h. Check [p, q.r, n) <- Nodiag [p, q, n), m = n+1, Check [p, r, m)
```

For simplicity we assume Nodiag is given by an explicit enumeration of the tuples satisfying the relation. This program is remarkable simple because the generation of possible solutions $[\text{Perm}]$ is written independant of the diagonal-test $[\text{Safe}]$, however with a strictly left to right execution order and naive backtracking, the behaviour will be poor. The program will generate all possible permutations. When Safe for example detects that the first and second queen are on the same diagonal, the program will only move the second queen after all configurations with another 3th, 4th, ... nth queen have been exhausted.

Rewriting the program such that parts of Safe are moved inside the permutation and the compatibility of the queens 1, ..., i is checked before the $i+1$ th queen is computed will make the program less transparent. { A more complex logic component cfr [6]}.

As we will illustrate, our intelligent backtracking will immediately react on a failure with moving the right queen. Using method a for determining the inputsets, the computation up to the point where Safe detects the first conflict is given in Fig. 2. The call $\text{Nodiag}[p_q, q_q, n_q)$ fails due to the values of p_q, q_q and n_q (the positions of queens 1 and 2). The substitutions causing the failures are generated by the steps [1) and [2). However [1) is in the inputset of [2) and [2) is the only possible culprit. The situation after backtracking is given in Fig. 3. A new position for the second and following queens has to be computed but part of the execution of Safe has been saved. The algorithm reacts on the detected failure with the replacement of one of the involved queens.

Remark that different calls are associated with the same computation step. This will be a general situation for logic programs because large parts are

Fig. 2. Execution of the program until $\text{Nodiag}(p_9, q_9, n_9)$. The bindings $p_9 \leftarrow 1$, $q_9 \leftarrow 2$, $n_9 \leftarrow 1$ derivable from the steps (0), (1) and (2) causes the elimination of all available definitions. Deep backtracking is necessary. Because (1) is in the inputset of (2), (2) is the only possible culprit. This is exactly the step placing the second queen.

Fig. 3 : The situation after deep backtracking. The elimination of procedure (c) of (2) is due to (1). The execution can take a new start. Remark that part of the execution of Safe has been saved.

determinate i.e. only one definition matching a call. This largely simplifies the task of manipulating the inputsets.

Notes

1. The behaviour still depends on the order in which the calls on Nodiag are generated. This can be avoided by pursuing the Safe computation until all failing calls are detected. Backtracking needs to remove all the found conflicts.
2. This example is typical for a large class of generate and test programs where the separation of the generate part and the test part gives a simple transparent logic, but, with naïve backtracking, an extremely inefficient behaviour.

6. Finding all solutions.

Up to now, we were only handling failures, however, at some point, a solution will be found and the user can be eager to know more solutions. A straightforward way to obtain another solution follows from the statement that the found solution must be rejected, in other words that there is a conflict with all calls composing the solution as set of suspects. The disadvantage of this approach is that nonessential dependencies are created between subproblems which are independent of each other. This slows down the further search process.

Using an example we will indicate how the information about the inputsets allows to find independent subproblems and to speed up the process of finding all solutions.

```
<- P(A.B.C.D.E.F.Nil, z)
a. P(1, p(x,y)) <- Split [1, l1, l2), Member [x, l1), Member [y, l2)
b. Split [Nil, Nil, Nil) <-
c. Split [x.Nil, x.Nil, Nil) <-
d. Split [x.y.l, x.l1, y.l2) <- Split [1, l1, l2)
e. Member [x, y.z) <- Member [x, z)
f. Member [x, x.y) <-
```

The derivation of a first solution is given in Fig. 4. The conflict induced by the rejection of that solution has steps [2) and [4) as possible culprits. Choosing [4) as culprit makes [4) dependent on [2). The systematic rejection of all solutions imposes step by step a total order over the steps of the execution. The computation degrades to the naïve backtracking algorithm and, although the subproblems Member [x, l₁) and Member [y, l₂) are independent of each other, we shall compute all solutions of Member [y, l₂) for each solution of Member [x, l₁). To avoid this duplication of effort on Member [y, l₂), we want to compute its solutions only once and to represent them as lemma's. We can use this lemmas to get a new set of solutions of the total problem for each new solution of Member [x, l₁)

In the remainder of this section, we will briefly outline when such savings are possible and feasible. We start with some definitions related to a solution of a problem.

A feasible set of subtrees of a solution is such that the steps occurring in these subtrees are not in the inputsets of steps outside the subtree [e.g. the subtrees with Member [x₁, z₁) and Member [y, l₂) as root but not the subtree with Split [1, l₁, l₂) as root). In other words, the set of calls in the roots

Fig. 4 : A first solution.

could have appeared in an intermediate goal statement leading to the solution.

A set of subtrees is nondeterminate if it contains a call with a nonempty set of available definitions.

The use of lemmas can be useful if we can distinguish two disjoint nondeterminate feasible set of subtrees [e.g. the subtree with Member $\{x, l_1\}$ as root and the subtree with Member $\{y, l_2\}$ as root). Then we isolate the subproblem defined by one set of subtrees and compute all its solutions. We represent each solution by a lemma and determine the validityset of the lemma's : all steps outside the subproblem which occur somewhere in the inputset of a step leading to the solution of the isolated subproblem [e.g. the solutions of Member $\{y, l_2\}$ are $y \leftarrow F$, $y \leftarrow D$ and $y \leftarrow B$, the validityset consists of step $\{0\}$).

The effort was worth while if the validityset still does not contain steps belonging to the solution of the other feasible nondeterminate set of subtrees. We then replace the roots of the set of subtrees by a single call whose solution is given by the first lemma and which has the other lemmas as available definitions. Then we reject the first solution and select the call using lemma's as culprit. This results in one solution for each lemma. [e.g. the solutions $z \leftarrow p(E, F)$, $z \leftarrow p(E, D)$ and $z \leftarrow p(E, B)$]. As next culprit we select a step from the other feasible nondeterminate set of subtrees. [e.g. step $\{2\}$]. In other words, we start by searching a new solution for the associated subproblem. Once found [e.g. $x \leftarrow C$] we again apply the lemma's to obtain a new set of

solutions [e.g. $z \leftarrow p(C, F)$, $z \leftarrow p(C, D)$ and $z \leftarrow p(C, B)$].

We have to remove the lemmas and to replace the call by the original set of calls once it is necessary to choose as culprit a step belonging to the validityset of the lemmas.

This approach gives the computation a flavour of parallelism where it is feasible i.e. where it is possible to distinguish independent subproblems each (eventually) having different solutions. In those situations this approach in fact makes the crossproduct of both solutionsets. The information in the inputsets allows to detect potentially independent subproblems.

7. Final remarks

We have presented a framework for purely syntactical analysis of the dependencies between subgoals. It shows the possibilities, but also the price to be paid, for a more intelligent behaviour which moves the burden of the control (algorithm = logic + control [6]) from the user to the execution mechanism.

It is important to realise that the reaction on a failure not only depends on the failure state but also on the history of previous failures.

The method of determining the inputsets is likely to depend on the purpose of the system. In the context of theorem proving, where the user is unable to handle control at all, one wants the most intelligent system which carefully analyses the state and the impact of the choice of a culprit on the size of the search space, whereas, for the execution of logic programs, it is more important to obtain a simple and fast execution mechanism using only a limited amount of space.

Our method can be considered a kind of 'truth maintenance' [4] for resolution theorem provers.

3. Acknowledgments.

This paper is extracted from [2]. For this research, which was supported by the Belgian 'Nationaal Fonds voor Wetenschappelijk Onderzoek', we are especially indebted to R. Kowalski and Y.D. Willems.

9. References

- [1] Bruynooghe, M.
Intelligent backtracking for an interpreter of Horn Clause logic programs.
Report CW 16, Afdeling Toegepaste Wiskunde en Programmatie,
K.U.Leuven, Belgium, september 1978.
- [2] Bruynooghe, M.
Naar een betere beheersing van de uitvoering van programma's in de
logika der Horn-uitdrukkingen. [In Dutch].
Doctoral dissertation, Afdeling Toegepaste Wiskunde en Programmatie,
K.U.Leuven, Belgium, may 1979.
- [3] Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R.
Un système de communication homme-machine en français.
Groupe d'Intelligence Artificielle, U.E.R. de Luminy,
Université d' Aix-Marseille, 1973.
- [4] Doyle, J.
A truth maintenance system.
Artificial Intelligence 12 (1979), 231-272.
- [5] Kowalski, R.A.
Predicate logic as a programming language.
Information Processing 74, North-Holland Pub. Co., Amsterdam,
1974, pp. 569-574.
- [6] Kowalski, R.A.
Algorithm = logic + control.
Comm. ACM 22, 7 (july 1979), 424-435.
- [7] Robinson, J.A.
Logic : form and function.
Edinburgh University Press 1979.
- [8] Sussman, G.J. and McDermott, D.V.
Why conniving is better than planning.
AI Memo 255A, MIT, april 1972.
- [9] Warren, D., Pereira, L.M. and Pereira, F.
PROLOG- The language and its implementation compared with LISP.
Proc. symp. on Art. Intell. and Programming Languages;
SIGPLAN Notices (ACM) 12, 8; SIGART Newsletters (ACM) 64
(august 1977), pp. 109-115.