PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

FINDING TEMPORARY TERMS IN PROLOG PROGRAMS

Pentti Vataja, Esko Ukkonen

Department of Computer Science
University of Helsinki
Tukholmankatu 2, SF-00250 Helsinki 25
Finland

ABSTRACT

We consider the memory management in backtrack-based Prolog implementations where the Prolog terms are represented as linked data structures that may have shared substructures. In any fixed activation of a clause P_0 <-- $P_1,P_2,\ldots,P_n$, we call a term t of the body $P_1,P_2,\ldots,P_n$ temporary if its representation at the end of the activation has no shared data structures with the terms of the clause which called P_0. Temporary terms can be utilized in garbage collection. Suppose, namely, that a temporary term t does not share data structures with the terms of the backtrack points in the search tree rooted at P_0. Then the memory space for t can be reclaimed immediately at the end of the activation of P_0 <-- $P_1,P_2,\ldots,P_n$. Two methods for discovering temporary terms are proposed. The dynamic method is applied together with the program execution. The static method is applied before the program is actually executed, but the outputs can be utilized in organizing garbage collection of every subsequent execution of the program, as long as the program does not change.

1 INTRODUCTION

A standard technique in Prolog implementations is to represent the Prolog terms as linked data structures that can have shared substructures (e.g. Warren 1977). When a data structure becomes unreachable during program execution the storage space reserved for it can be reclaimed and used for storing new data structures. Finding the unreachable storage areas is the well-known garbage collection problem for linked structures.

The garbage collection is normally done in an indirect way: the collector knows the terms which are still needed in program execution ("the live variables") and marks all data structures representing them. The storage areas which remain unmarked are known to be unreachable. The whole storage is finally traversed and the unmarked areas are reclaimed. The use of this method in Prolog implementations has been considered e.g. by Bruynooghe (1984).

To complement this approach we examine in this paper an opposite idea which could be called **direct** garbage collection in Prolog. Our goal is to develop methods for locating at any moment of the Prolog program execution the terms which are not needed any more and which do not share substructures with terms that are still needed. Obviously, the storage space reserved for such terms can be reclaimed immediately. Opposite to the indirect method, the reclaming can be done without traversing through the whole storage reserved for terms.

In the next section we define temporary terms. This is the key-concept for the rest of the paper. The remaining sections give two methods for finding such terms. A direct garbage collection can be based on the results supplied by these methods.

Our methods will be developed in the general framework of backtrack-based Prolog implementations (e.g. Warren 1977, van Emden 1984) whose main principles the reader is supposed to be familiar with. Of course, we also assume basic knowledge on logic programming and Prolog (e.g. Kowalski (1974), Clocksin & Mellish (1984)).

2. TEMPORARY TERMS

Let

(1) $P_0 \text{ <-- } P_1,P_2,\ldots,P_n$

be a **clause** of a Prolog program, with **head** P_0 and **body** $P_1,P_2,\ldots,P_n$. Each **predicate** P_j, $0<=j<=n$, is of the form $P_j(r_1,\ldots,r_m)$ where the $r_1,r_2,\ldots,r_m$ are the **terms** of P_j. Each term is either a variable or an expression of the form $f(g_1,\ldots,g_k)$ where f is a function name and $g_1,\ldots,g_k$ are again terms.

This means, in particular, that each occurrence of a variable in (1) is a subterm of some term of some predicate P_j. We call these subterms (i.e., subterms that consist of only a variable name) the **variable terms** of (1). For example, the variable terms of clause

```
split(H,c(A,X),c(A,Y),Z)    <--
              order(A,H), split(H,X,Y,Z)
```

are, from left to right, H,A,X,A,Y,Z in the head and A,H,H,X,Y,Z in the body. Intuitively, we use variable terms as pointers to the places in the program text that in unifications can be substituted by terms represented as linked data structures. Suppose, therefore, that it is dynamically or statically possible to distinguish whether a variable refers to a linked structure or to a data element stored outside the storage area for linked structures. (Statically this can be achieved in Prolog dialects with explicit type declarations; e.g. Nilsson (1983), Mycroft & O'Keefe (1983).) Whenever this separation is possible, we do **not** include into variable terms the variables referring to data. In the above example, if H and A are known to refer to data, the variable terms by our definition are X,Y,Z in the head and X,Y,Z in the body.

Consider then **instances** of (1), that is, clauses that are obtained from (1) by some substitution for variables. The variable terms of such an instance are the subterms that correspond to the variable terms of (1).

Temporary terms are a subset of variable terms, to be defined as follows. Let t be a variable term in the body of clause (1). Consider some fixed activation of (1) during the execution of the program. The activation starts with unifying the head of (1) with the current subgoal and creating the data structures that represent the unified forms of the terms of (1). So also the data structure for t is initialized. Initialization often just means creating a link pointing to older structures, which thus establishes a shared substructure. Variable term t is called **isolated** at this moment if the data structure representing t does not have shared substructures with the variable terms of P_0. Note here that shared program text ("structure sharing") or shared primary data is not considered as a shared substructure. If the data structure for t is still isolated when the activation of (1) becomes complete after successfully solving the last subgoal P_n, then we call t a **temporary term** in this activation of (1).

Obviously, when the activation of (1) is completed the data structures for a temporary t are disjoint from the data structures for the variable terms of head P_0 (and hence disjoint from structures for the terms of P_0). A temporary term is needed only for the "local" purposes of (1). Since t can have shared substructures with the terms of the clause which called (1) only if it has shared substructures with the terms of P_0, we actually have:

Proposition 1. Let t be a temporary term in an activation of (1). Then t does not share substructures with the terms of the head P_0 in this activation or with the terms of predicate activations that in the backtrack search tree are outside the subtree rooted at P_0.

Being a temporary term is a dynamic property: It is possible that a variable term t is not temporary in all activations of (1). If t is always temporary then it is called a **statically temporary term** of (1).

Proposition 1 implies that after accomplishing the activation of (1), the storage space reserved for a temporary term t can be reclaimed, provided that t does not share substructures with some term of a backtrack point ("nondeterministic node") which is in the subtree rooted at P_0. Hence for reclaming it suffices to know that there are no backtrack points in the subtree. In Prolog dialects with the "cut" this information is often immediately available. Also implementations of Prolog without "cut" are easily modified to supply the information but in this case, unfortunately, the requirement that there cannot be any backtrack points in the subtree rooted at P_0 seems rather strong and prevents finding terms that actually are garbage. However, a more careful analysis is possible to find more storage areas that can be reclaimed.

Let us say that a temporary term t of an activation of (1) is **strongly temporary** if t does not share substructures with any term of a backtrack point in the subtree rooted at P_0. Clearly, the storage space reserved for a strongly temporary t can be freed after accomplishing the activation of (1).

In this paper we mainly concentrate on finding temporary terms and just comment on the often rather obvious changes needed to find strongly temporary terms. Section 3 deals with the dynamic case and section 4 sharpens the analysis to the static case.

3 DYNAMIC ANALYSIS

The dynamic analysis is performed together with the program execution. As a result the analysis announces the temporary terms of all activations of clauses. The method to be presented is approximative: every term claimed temporary by the method really is temporary but some temporary terms may remain undiscovered. The method can be used provided that the underlying Prolog implementation allows to separate during the program execution the data terms from terms represented as linked data structures.

Consider again a fixed activation of a clause (1). If the data structures representing two variable terms t and t' of (1) in this activation have a shared substructure, we write t D t'. Relation D is called the **dependency relation** of the activation. One now immediately notices that a variable term t of the body of (1) is temporary if and only if there is no variable term t' in the head such that t' D t.

Of course, D is a symmetric relation: t D t' if and only if t' D t. The relation is not transitive since, obviously, t D t' and t' D t'' does not imply that t D t''. However, we will compute (an approximation of) D as if it were transitive. This means, as already mentioned, that the method does not necessarily find all temporary terms.

The computation of relation D can be based on some basic dependencies between the variable terms. Let t and t' be two variable terms of P_0 <-- $P_1,\ldots,P_n$ such that at least one of them belongs to the body. Then we write t BD t' if the variable name in t is the same as the variable name in t'. Relation BD is easy to compute for each clause statically from the program text.

The corresponding relation between the variable terms of the head P_0, is defined dynamically: Let t and t' be two such terms. Assume moreover that the variable name in t equals the name in t'. At the beginning of the activation of P_0 <-- $P_1,\ldots,P_n$ we are considering, head P_0 is unified with the current subgoal. The unification may create a link between the structures representing t and t'. In this case (it is reasonable to assume that the unifier is able to supply this information) we write t UD t'.

The rest of the computation of D simply propagates the transitive effect of relation BD U UD to all activations of clauses during the execution of the Prolog program. Formulated inductively, the method for computing D for a fixed activation of a clause P_0 <-- $P_1,\ldots,P_n$ proceeds as follows:

D1. If n = 0 (i.e., the clause has empty body), relation D is the transitive closure of the relation UD for this activation of P_0 <-- . Moreover, D is defined to be the **propagated dependency relation** of this activation.

D2. Let n > 0. Assume that the activation of P_0 <-- $P_1,P_2,\ldots,P_n$ has been successfully completed and that the subgoals $P_1,P_2,\ldots,P_n$ were solved by unifying P_i with a clause Q_i <-- z_i, 1 <= i <= n. Also assume that the propagated dependency relation of the corresponding activation of each Q_i <-- z_i is PD_i. Now propagate each relation PD_i to the activation of P_0 <-- $P_1,P_2,\ldots,P_n$ by setting t PD t' for all variable terms t and t' of P_i such that r PD_i r' where r and r' are the variable terms of Q_i, that correspond to t and t'. So we have three relations among the variable terms of P_0 <-- $P_1,P_2,\ldots,P_n$: the relation PD and the static relation BD and the unification based relation UD. Relation D is then computed as the transitive closure of combined relation PD U BD U UD. Moreover, the propagated dependency relation of this activation of P_0 <-- $P_1,P_2,\ldots,P_n$ is D restricted to the variable terms of P_0.

The detailed implementation of this method should be quite obvious: to every activation of a clause one must attach a data structure that first represents relation BD U UD and then collects during the execution of the program the propagated relations PD_i. When the activation becomes completed, the transitive closure of the collected relation has to be computed.

The relation between temporary terms and the computed relation D can be summarized as follows:

Proposition 2. Variable term t is temporary in an activation of clause P_0 <-- $P_1,P_2,\ldots,P_n$ if P_0 has no variable term t' such that t D t' where D is the approximate dependency relation of the activation, as computed by the above method.

After minor modifications the above method also finds strongly temporary terms. A strongly temporary term of P_0 <-- $P_1,P_2,\ldots,P_n$ is not allowed to share substructures with the terms of the backtrack points in the tree rooted at P_0. It is natural to transmit information on such shared structures together with the propagated dependency relation. Let Q_i <-- z_i and PD_i, $1 <= i <= n$, be as in step D2. We now assume that together with PD_i there follows a list of those variable terms of Q_i that share substructures with some term of a backtrack point in the tree rooted at Q_i. The variable terms of P_i that correspond to the listed variable terms of Q_i are marked. Then D as well as the propagated dependency relation of this activation of P_0 <-- $P_1,P_2,\ldots,P_n$ is computed as in step D2. The list of variable terms to be transmitted together with the propagated relation contains all the variable terms of P_0 if P_0 is a backtrack point. Otherwise the list contains all terms t such that t D t' for some marked variable term t' of the body $P_1,P_2,\ldots,P_n$.

Finally, it should be clear that a variable term t is strongly temporary in an activation of clause P_0 <-- $P_1,P_2,\ldots,P_n$ if t' D t for no variable term t' in P_0 and for no marked variable term t' in $P_1,P_2,\ldots,P_n$.

4 STATIC ANALYSIS

4.1 The method

The purpose of the static analysis is to find the statically temporary terms of a given Prolog program. Recall that a variable term of a clause is statically temporary if it is temporary in all activations of the clause which are possible during any execution of the program, with a fixed top goal but with varying data. While the dynamic method is applied during every program execution, it suffices to apply the static method only once but the results can be utilized in the direct carbage collection of every subsequent execution of the program, as long as the program does not change.

Basically, the static method involves a truncated simulation of all unification patterns possible during program execution. Because of recursion, a finite and accurate simulation of all patterns is impossible. Therefore we use a truncation technique called the **depth abstraction.** This again leads to an approximate solution: Some statically temporary terms may remain undiscovered. To get reasonably accurate results it is necessary that - unlike in pure Prolog - a static separation of data from structures is possible (for example, explicit type declarations are needed). The accuracy still improves if the ground terms of the predicates are known, see Warren (1977).

To trace the backtracking, we use as technical tools the dotted "items" widely applied in the theory of context-free parsing (Aho and Ullman 1977). Similar techniques in analyzing Prolog has been used earlier by Sato and Tamaki (1983).

Before giving the algorithm we define some preliminary constructions. To propagate information on shared substructures we augment the program with **color variables.** Within every clause, each variable as well as each occurrence of a function name gets its own color variable. However, such variables are associated only with function names and variables that refer to linked data structures. Hence a color variable is not associated with a variable that refers to primary data nor with a constant whose value is represented by an empty link (e.g. "nil"). For example, the clause

```
split(H,c(A,X),c(A,Y),Z)     <--
          order(A,H), split(H,X,Y,Z),
```

where H and A are assumed not to refer to structures, gets, with the color variables shown as superscripts, the form

$$\text{split}(H,c^{R1}(A,X^{R2}),c^{R3}(A,Y^{R4}),Z^{R5}) \text{ <-- } \text{order}(A,H),\ \text{split}(H,X^{R2},Y^{R4},Z^{R5}).$$

The color variables are used to indicate shared substructures: If two entities get the same color variable their representations can have shared substructures. It is assumed in the sequel that the color variables are associated with our Prolog program in a proper way.

The **depth** of a term is defined as the maximum number of nested function names in the term. The **level** of a subterm is the number of nested function names around it. Let t be a term with color variables and k an integer. Then the **depth k abstraction** of t, denoted (t)k,is a term of depth k, formed by replacing every level k subterm with a new variable. A color variable is associated with the new variable only if the replaced subterm contains at least one color variable. If the color variables in the subterm are $R_1,\dots,R_s$, then one of them, say R_1, becomes to the color variable of the new variable, and all occurrences of variables $R_2,\dots,R_s$ outside the subterm are replaced by R_1. In an obvious way, depth k abstraction can be applied on abstracted terms, clauses, and also to items, to be defined later on.

Consider as an example a term t =

$$f^{R1}(g^{R2}(X^{R3},g^{R4}(X^{R3},L^{R5})),c^{R6}(L^{R5}))$$

of depth 3. Then

$$(t)2 = f^{R1}(g^{R2}(X^{R3},T1^{R3}),c^{R6}(L^{R3}))$$

and $((t)2)1 = f^{R1}(T2^{R2})$, where T1 and T2 are the new variables.

A Horn clause including a dot in the body is called an **item.** An item is allowed to contain color variables. An item of the form P <-- .z is an **initial item** and an item of the form P <-- z. is a **closed item.** An item is called a **k-item** if it contains only terms of depth at most k.

If an expression E such as clause or item is identical to another expression F or differs from F only in the variable names, E and F are called **variants** of each other, denoted as E = F (modulo renaming). This applies to sets of expressions, too.

Let S be a program with initial goal and with color variables, and let k be an integer. The set I_k of k-items for S consists of all k-items that are formed from some clause of S by applying substitutions and depth reductions.

Next we define two binary relations in I_k, the relations DOWN and NEXT. Unification in these definitions is applied also to the color variables, such that the substitutes for these variables are again appropiate color variables.

Let u = P <-- x.Xy be in I_k where X denotes the first predicate after the dot. Then we write u DOWN v for all v in I_k such that v is $((R\text{<--}.z)\Theta)k$. Here R<--z is a clause of S such that R is unifiable with X and Θ is the corresponding most general unifier.

Similarly, if u = P <-- x.Xy then we write u NEXT v for all items v in I_k that satisfy (a) and (b):

(a) There are items $w_1,w_2,\dots,w_r$, 1 <= r, such that u DOWN w_1 and w_1 NEXT w_2, ... , w_{r-1} NEXT w_r, and w_r is a closed item;

(b) From (a) it follows (the proof is by induction) that w_r must be of the form R <-- z. where R is unifiable with X. Let Θ be their most general unifier. Then v = ((P<--xX.y)Θ)k.

The item set construction for program S can now be completed. Let <-- .z be the initial goal of S. The k-item (<-- .z)k is called the initial k-item for S. The **k-item set for program S and goal** z is the finite set $I_k(S,z)$ =

$$\{ y \in I_k \mid (\texttt{<-- .z})k \ (\text{DOWN} \cup \text{NEXT})^* \ y \}$$

where (DOWN ∪ NEXT)* denotes the transitive closure of relation NEXT ∪ DOWN.

The construction of set $I_k(S,z)$ is almost equivalent to a similar construction by Sato and Tamaki (1983). The main difference is that we have color variables in items and the substitutions Θ are assumed to have been applied on a proper way on the color variables, too.

We skip a description of the (more or less obvious) relationship between $I_k(S,z)$ and the program execution to solve the goal <-- z. We turn to the problem of finding the statically temporary terms with the help of set $I_k(S,z)$. For each closed item in $I_k(S,z)$ we first define temporary subterms for items. This is analogous to the use of relation D in section 3. Here the relation is encoded in the color variables of entities and therefore no explicit transitive closure calculations are needed after forming $I_k(S,z)$.

Let u = P <-- x. be a closed item in $I_k(S,z)$, and let t be a subterm in the body x. Term t is called a **temporary subterm** of u if no color variable has an occurrence both in t and in the head P of u.

The temporary subterms of an item have the following connection to the temporaryness of subterms in actual clause activations during program execution:

Proposition 3. Let the execution of S successfully activate a clause R <-- y, with solution substitution Θ, and let u be a closed item $I_k(S,z)$ such that u = ((R<--y.)Θ)k (modulo renaming). If t is a temporary subterm of item u, then the subterm of (R<--y)Θ that corresponds to t, is temporary in the program execution.

Proposition 3 simply says that temporary subterms of an item u indicate temporary subterms in all activations whose unified clause (R<--y)Θ can be abstracted to u. Since any unified clause can be abstracted to some item in $I_k(S,z)$, statically temporary terms of clause R <-- y in S can be found by considering simultaneously all the closed items that are instances of R <-- y. In this way we obtain :

Proposition 4. Let t be a variable term in the body of a clause R <-- y of S. Term t is statically temporary if in all closed items in $I_k(S,z)$ that are instances of R <-- y, the subterms corresponding to t are temporary subterms.

A question remains: What is a suitable k? Obviously, the larger is k, the larger is the set of statically temporary terms that can be found using the method of Proposition 4. On the other hand, the size of set $I_k(S,z)$ increases rapidly with k. It seems reasonable to use k which is at least as large as the largest depth of a term in S, but further research is needed with the problem.

4.2 Example

Let us apply the static method on the Quicksort program of Clocksin and Mellish (1981):

1. split(H,nil,nil,nil) <--

2. split(H,c^{R1}(A,X^{R2}),c^{R3}(A,Y^{R4}),Z^{R5})

    <--order(A,H),split(H,X^{R2},Y^{R4},Z^{R5})

3. split(H,c^{R1}(A,X^{R2}),Y^{R3},c^{R4}(A,Z^{R5}))

    <--order(H,A),split(H,X^{R1},Y^{R3},Z^{R5})

4. append(nil,L^{R1},L^{R2}) <--

5. append(c^{R1}(X,L^{R2}),M^{R3},c^{R4}(X,N^{R5}))

    <-- append(L^{R2},M^{R3},N^{R5})

6. qsort(nil,nil) <--

7. qsort(c^{R1}(H,T^{R2}),S^{R3})

    <-- split(H,T^{R2},A^{R4},B^{R5}),

      qsort(A^{R4},$A1^{R6}$),qsort(B^{R4},$B1^{R7}$),

      append($A1^{R6}$,c^{R8}(H,$B1^{R7}$),S^{R3})

Assume that (by a type declaration not given here) variables appearing as the first argument of functors "split" and "c" are known not to refer to a data structure while all other variables refer to such structures. Then the variable terms of clause 7 are t1 = T, t2 = S, t3 = T, t4 = A, t5 = B, t6 = A, t7 = A1, t8 = B, t9 = B1, t10 =A1, t11 = B1, t12 = S. When I_k(S,qsort(Y,X)) is formed for this program with k = 1, we obtain, among others, a closed item

$$\mathrm{qsort}(c^{R1}(H,Q^{R2}),c^{R3}(B,W^{R4}))\ \texttt{<--}$$

$$\mathrm{sp}(H,c^{R2}(A,R^{R2}),c^{P1}(A,S^{P2}),c^{P3}(Y,T^{P4}))$$

$$\mathrm{qsort}(c^{P1}(A,S^{P2}),c^{G1}(B,U^{G2})),$$

$$\mathrm{qsort}(c^{P3}(Y,T^{P4}),c^{R4}(D,V^{R4})),$$

$$\mathrm{ap}(c^{G1}(B,U^{G2}),c^{K3}(H,X^{R4}),c^{R3}(B,W^{R4})).$$

This is an instance of clause 7 where Q,W,R,S,T,U,V,W,X are variable names created in depth abstraction.

The subterm that in this item corresponds to variable term t3, is $c^{R2}(A,R^{R2})$. Since color R2 appears in the head of the item, the subterm is not temporary. Hence t3 cannot be statically temporary. Similarly, the subterm that corresponds to t4 is $c^{P1}(A,S^{P2})$. Colors P1, P2 do not appear in the head. This subterm is temporary. Hence the static temporaryness of t4 is not forbidden by this item. Conforming in the same fashion, one sees that the subterms for t4-t8, t10 are temporary while subterms for t3, t9, t11, and t12 are not.

By examining all the closed items that are instances of clause 7 (we skip the details), it turns out that the subterms corresponding to t4, t5 and t7 are always temporary. Hence we have found that t4, t5, and t7 are statically temporary.

The other variable terms remain non-temporary by the static analysis. A careful examination of the program reveals, however, that there are no other statically temporary terms than terms t4, t5, t7 found above. Hence our approximate method works accurately in this example.

Obviously, terms t4, t5, and t7 are statically temporary in the strong sense if there is a "cut" at the end of clause 7 (or "cut" is appropriately used in the other clauses). Another possibility to discover strong temporaryness is to collect information on backtrack points using a dynamic method, as delineated at the end of Section 3.

5 CONCLUSION

A rather informal description of two methods useful in organizing memory management in Prolog implementations were given, based on the concept of a temporary term. Further research, both experimental and theoretical, is needed to test the practical value of the proposed techniques. A particularly interesting problem for such a study is the applicability of our ideas in parallel implementations.

REFERENCES

Aho, A.V. and Ullman, J.D.
Principles of Compiler Design.
Addison-Wesley, 1977.

Bruynooghe, M.
Garbage collection in Prolog interpreters. In: J.A. Campbell (ed.), Implementations of Prolog. Ellis Horwood, 1984, 255-267.

Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1981.

van Emden, M.H.
An interpreting algorithm for Prolog programs. In: J.A. Campbell (ed.), Implementations of Prolog. Ellis Horwood, 1984, 93-110.

Kowalski, R.A.
Predicate logic as programming language. In: Proc. IFIP-74 Congress. North-Holland Publishing Company,1974, 569-574.

Mycroft,A. and O'Keefe, R.A.
A polymorphic type system for Prolog. Artificial Intelligence 23(1984), 295-307.

Nilsson, J.F.
On the compilation of Domain-Based PROLOG. In: Proc. IFIP-83 Congress. North-Holland Publishing Company, 1983, 293-298.

Sato, T. and Tamaki, H.
Enumeration of success patterns in logic programs. In: Automata, Languages and Programming, 10th Colloquium, Barcelona 1983. Lecture Notes in Computer Science 154, Springer-Verlag, 1983, 640-652.

Warren, D.H.D.
Implementing Prolog. Report 39,40. Department of Artificial Intelligence. University of Edinburgh, 1977.