SELECTIVE BACKTRACKING FOR LOGIC PROGRAMS

Luís Moniz Pereira
António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa, Portugal

Abstract
We present a selective backtracking method for Horn clause programs, as applied to Prolog [2][6][11][12], a programming language based on first-order predicate calculus [3][4], developed at the university of Marseille [10].
This method is based on the general method expounded in [7] for backtracking intelligently in AND/OR trees. It consists, essentially, in avoiding backtracking to any goal whose alternative solutions cannot possibly prevent the repetition of the failures which caused backtracking.
This is a renewed version of an earlier report [8], which was spurred by the work of Bruynooghe [1].
In [9] we present an implementation of a selective backtracking interpreter using the methods discussed in this paper.

1. Introduction

The desire for control over resolution theorem provers [5] has always flourished. General search strategies, however, have remained too general. On the other hand, the standard backtracking methods remain plagued by the inefficiency of looking for remedy where it cannot be found.

This paper provides a strategy for backtracking selectively in top-down executions of Horn clause programs, which avoids searching irrelevant branches of the corresponding AND/OR tree. It consists, essentially, in avoiding backtracking to any goal whose alternative solutions cannot possibly prevent the repetition of the failures which caused backtracking.

This more intelligent backtracking strategy is based on the general method expounded first in [7] for backtracking intelligently in AND/OR trees, and is presented here as applied to Prolog [2][6][11][12], a programming language based on first-order predicate calculus [3][4], developed at the university of Marseille [10]. The adaptation of the techniques developed here to resolution-logic theorem provers is, we believe, straightforward, if its executions can be modelled by an AND/OR tree.

This paper is a renewed version of an earlier report [8], that was spurred by the work of Bruynooghe [1], although improving it in several respects.

Our method has been tested in its implemented form. The implementation, which further elucidates our strategy, is presented in [9].

2. Prolog AND/OR trees

To solve a goal a clause must be found whose head matches the goal and whose body goals (if any) can all be solved. Execution of a goal generates an AND/OR tree with that goal as the root.

The clauses that match a goal give rise to OR branches at that goal. Each such OR branch is split into AND branches. One such branch leads to the node where unification between the goal and the clause head is achieved. Each remaining AND branch leads to a goal in the body of the clause. The unification node is itself an AND of the unifications of corresponding arguments of the goal and head, where each argument may be a compound term giving rise to further AND branching.

3. Backtracking selectively in Prolog AND/OR trees

In the execution of a Prolog program, a goal G is said to fail when G is selected for activation and there are no alternative clauses that G can match.

Standard backtracking consists in going back to reactivate the goal activated just before G.

Selective backtracking consists in updating the current set of goals selected as backtrack goals, by analysing the reasons for the failure of G, going back to reactivate the most recently activated goal present in that set, and deleting it from the set.

Whenever an activated goal G fails, the following are the sole goals selected for inclusion in the set of backtrack goals - these are the only goals that may prevent repetition of the same failure:

- The parent goal of G, since an alternative solution for it will avoid reactivation of G. It is an avoiding goal for G.

- All goals after whose reactivation the arguments of G may be modified so as to allow G to match some clause head which it did not match in its current form. These are the modifying goals for G.

Why are these the only goals selected for backtracking when activation of G has failed?

First, the goals that necessarily avoid G are its ancestors. Selecting only the parent of G is, however, sufficient for selecting all its ancestors, because if and when the parent fails its own parent will be selected as a backtrack goal, and so on, up to the top goal if necessary.

Second, let us discuss which are the only goals that may prevent G from failing again. Since G failed, for every clause for G either its head did not match G or some goal S subsequent to G failed. Backtrack goals that may prevent S from failing again have assumedly been selected already, when S failed. So the only possibly new backtrack goals that may prevent G from failing are those after whose reactivation the arguments of G may be modified so as to allow G to match some clause head that did not match the current failed G. If G fails on first activation it means no matching clause was found; in this case the parent goal and the modifying goals for G are clearly the only backtrack goals that may allow solution of the top goal. This is the basis for the recursive argument given above.

How are the modifying goals for G obtained?

The modifying goals for G are all those selected at each failed attempted match of G with a clause head, because each clause provides an independent possibility for solving G.

This is the OR rule.

For each failed match, since unification is an AND of several matching conditions, every individual condition that failed must be modified to allow unification to succeed. For every individual failed condition there is a (possibly empty) set of modifying goals. To allow modification of all failed conditions backtracking must occur up to the least recent among the most recent goals in each set (if one set is empty no modifying goals exist for a failed condition, and thus for the whole match). Because there may be no alternatives left at that least recent goal, it may not be able to modify the failed conditions depending on it. Thus, the least recent of the most recent remaining modifying goals for those conditions must also be selected for backtracking (again, if no modifying goals remain for one of those conditions, no more goals should be selected). This argument may be repeated until some condition has no more modifying goals. No modifying goals for any other condition should be considered: if some condition cannot be modified the match will always fail, regardless of modifications of other conditions; on the other hand, if one or more conditions are modified but some others are not, a new failure of the goal G will then select a set of modifying goals for any remaining failed conditions.

So, the set of modifying goals selected for a failed attempted match is the least of the sets of modifying goals for individual failed conditions, according to the following (lexicographic) order among the sets:

- The empty set is the least of all sets.

- If two sets have different most recent goals, the least set is the one with the least recent of them.

- The order between two sets with equal most recent goals is the order between those same sets with the most recent goal deleted from them.

This is the AND rule.

What are the modifying goals for a failed matching condition?

Since every failed condition is a unification conflict between two different constant names (ie. constants or functors), the modifying goals for the conflict are the modifying goals for each constant name, because changing either of them may solve the conflict.

The modifying goals for a conflicting constant name are readily obtained if, during execution, each individual binding of a constant name in a goal is tagged with its set of modifying goals (a detailed analysis of all the various binding possibilities is carried out in a subsequent section). In such taggings the reference to a goal may be its number, goals being numbered in the order they are activated. On backtracking, both tagging and goal numbering are undone.

Because any goal is an avoiding goal for its subgoals (since it is their ancestor), no goal is a modifying goal for its subgoals. Thus tagging should only be performed when a goal is solved, not during unification of a goal with a clause head.

Finally, a more expediently implemented but less precise selective backtracking can be obtained if, for each failed goal, no analysis is made of which are the failed conditions in each failed match. In that case, all modifying goals for every goal term are selected as backtrack goals, ie. failure is not associated with any particular goal term but with all of them.

4. On the possible types of unification conflicts

In this section we concentrate on the different types of conflict that may cause failure of the matching of a goal with a clause head.

We distinguish two cases.

First case. One of the constant names textually occurs at the conflicting position in the clause head.

e.g. (1) ←p(a)
p(b)←

(2) ←p(X) { X=...=a }
p(b)← (X bound to a)

(3) ←p(X,X)
p(a,b)←

For the match to succeed the other constant name must change. If this other constant name is also textually present in the goal, as in (1), no modifying goals exist for the conflict, meaning that it cannot be solved. If the second constant name is not textual, then it is obtained through some variable, as in (2). In this case, the modifying goals are those for the binding of the variable. When, in particular, the second constant name, binding a variable, is textually present in the clause head and is transmitted to the conflicting position through a multiple occurrence of that goal variable, as in (3), the conflict is irrevocable.

Second case. No constant name textually occurs at the conflicting position in the clause head.

There is a variable at that position, which must occur somewhere else in the clause head to receive the mismatching constant name from the goal itself, either textually or by a bound variable.

e.g. (1) ←p(a,b)
p(X,X)←

(2) ←p(a,Y) { Y=...=b }
p(X,X)←

(3) ←p(Z,Y) { Z=...=a }
p(X,X)← { Y=...=b }

(4) ←p(Y,Y,b)
p(a,X,X)←

In this case, the conflict is really between two constant names referred to by the goal. Accordingly, the conflict may be solved if either of them changes, and the modifying goals are the ones for each constant name. If, in particular, both conflicting constant names are textually present in the goal or head, as in (1) and (4), the conflict is irrevocable.

In all cases, a conflict is irrevocable if both conflicting constant names are

textually present in the unification, and it is revocable if at least one conflicting constant name is not textually present in the unification.

Irrevocable conflicts can only be avoided, whereas revocable conflicts can either be avoided or modified by the undoing or redoing of a binding.

5. On the dependency of bindings on goal matches

We next examine how the presence of a constant name in the binding of a variable depends on goal matches.

The presence of a constant name as part of the binding of a textual variable in a goal G depends solely on the goals whose matches have transmitted that part of the binding. All of these, except the ancestors of G, are modifying goals for that constant name present in G.

1) If in a match a variable is bound to some non-variable term directly, ie. without any intervening variables, then all the constant names part of that binding will be tagged with the matching goal, when it is solved.

e.g.
```
←p(X)
 p(f(Y,a))←q(Y)        'f' and 'a' will be tagged with goal 'p'
```

2) The remaining case is where a variable becomes dependent, by unification, on other variables, even though the latter may have not yet acquired a value.

2.1) The simplest subcase is when a variable occurring in the clause head becomes dependent on the actual or future binding (possibly in the same match) of a matching variable in the goal.

Because the goal is only an avoiding goal for any failed goal in the body of the matching clause, reference to the goal should not appear in the tagging of any constant name binding to those variables.

e.g.
```
(1)  ←p(X)          { X=...=a }      (2)  ←p(X,X)
      p(Y)←q(Y)                           p(a,Y)←q(Y)

(3)  ←p(X,X)                         In all examples, reference to goal 'p'
      p(Z,Y)←r(Z),q(Y)               will not appear in the tagging of 'a'
             r(a)←                   bound to Y.
```

The particular case, as in (3), where two variables in the clause head are unified through the same multiple occurring variable in the goal is already catered for.

2.2) Another subcase is where a goal variable becomes dependent on some future binding (possibly in the same match) of a matching variable in the head.

2.2.1) If there is a single occurrence of the variable in the head, then the future binding can only be made in the execution of a subgoal. Any failure due to the transmission of that binding, via the goal variable, will select that subgoal for backtracking. But that subgoal, through its parent, will reactivate all its ancestors, including the current goal, if need be. Consequently, the dependency of the goal variable on the current goal needs no tagging.

e.g.

```
←p(X)
 p(Y)←r(Y)
      r(a)←
```

reference to goal 'p' will not appear in the tagging of 'a' bound to X.

2.2.2) If there is a multiple occurrence of the head variable, it is linking two or more terms in the goal.

2.2.2.1) If both terms are ground no tagging occurs.

e.g.

```
←p(X,a)      { X=...=a }
 p(Y,Y)←
```

2.2.2.2) If a constant name in one of the terms is bound to some variable in the other term it will be tagged with the goal, in that binding.

e.g.

```
←p(X,a)
 p(Y,Y)←
```

2.2.2.3) If a binding is between two free variables but, upon solution of the goal, they have become bound to some non-variable term by a subgoal, then reference to the goal should not be put in the tagging of that term, which already refers to the subgoal.

e.g.

```
←p(X,Y)
 p(Z,Z)←q(Z)
        q(a)←
```

reference to goal 'p' will not be put in the tagging of 'a' bound to X or Y.

On the other hand, if both variables are still free after the goal is solved, any subsequent binding of a constant name to them will have to be tagged with a reference to the goal. This is easily achieved if both variables are tagged with a reference to the goal.

e.g.

```
←p(X,Y)
 p(Z,Z)←
```

reference to goal 'p' will be tagged to X and Y.

To sum up:

Because parent goals of failed goals are always selected as backtrack goals, the goal dependencies created by simple transmission of bindings up and/or down the tree (through chains of ancestors possibly linked by common variables at brother goals) should not be noted explicitly.

Any solved goal in whose match a goal variable was bound to a textual non-variable term must be retained as a modifying goal for any constant name which then became part of the binding of that variable.

The only other case in which a solved goal must also be retained as a modifying goal is when two (or more) still free variables in the goal have been unified to one another in the matching of the goal.

6. Selective backtracking examples

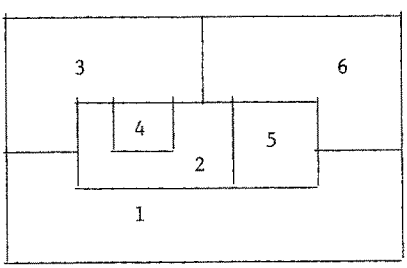
6.1 Map colouring example

This example is a program for colouring any planar map with at most four colours, such that no two adjacent regions have the same colour (proved to be always possible).

The program consists of a complete list of pairs of different colours, taken from a collection of four. These constitute the admissible pairs of colours for regions next to each other.

```
next(blue,yellow)←
next(blue,red)←
next(blue,green)←
next(yellow,blue)←
next(yellow,red)←
next(yellow,green)←
next(red,blue)←
next(red,yellow)←
next(red,green)←
next(green,blue)←
next(green,yellow)←
next(green,red)←
```

To obtain a colouring of the map below, we give as a goal to the program all the pairs of regions that are next to each other. This can be done systematically by first pairing region 1 with higher numbered regions next to it, then region 2, etc.




```
goal(R1,R2,R3,R4,R5,R6)←next(R1,R2),next(R1,R3),next(R1,R5),next(R1,R6),
                         next(R2,R3),next(R2,R4),next(R2,R5),next(R2,R6),
                         next(R3,R4),next(R3,R6),
                         next(R5,R6)
```

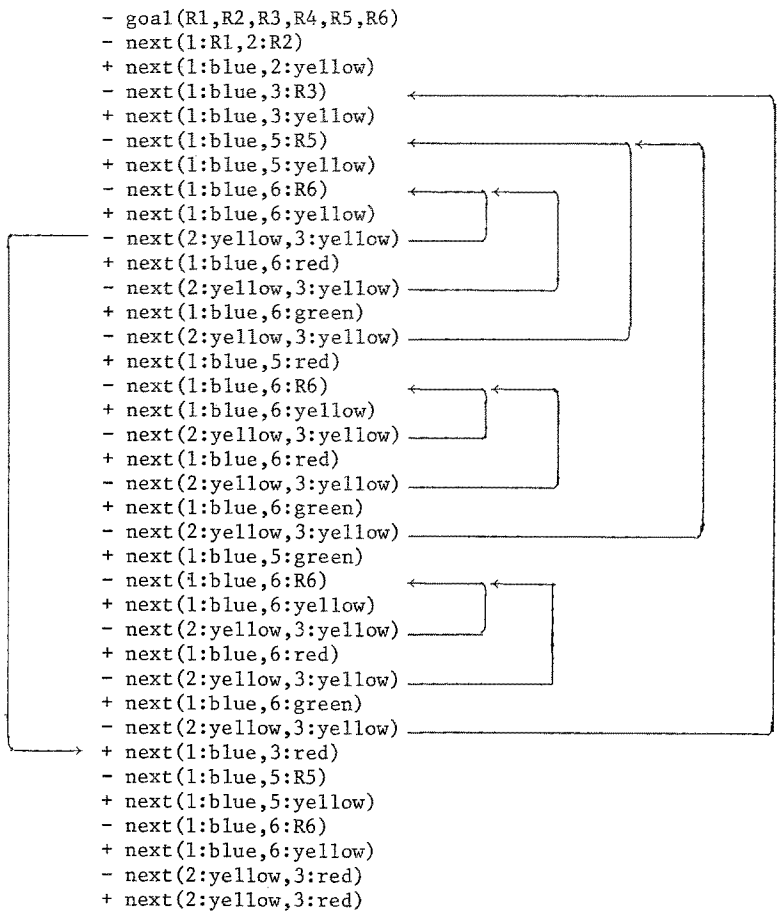
The effect of selective backtracking is best seen in the partial trace below of an execution using standard backtracking.

To help to follow the trace we have replaced the calls in the goal, which are of the form next(R_i,R_j), by calls of the form next(i:R_i,j:R_j).

In the trace, each procedure call is displayed with the current value of its arguments, and is preceded by a '-' sign; each time execution of a goal is successfully completed the call is displayed again, with the new current values of its arguments, this time preceded by a '+' sign.

On the right-hand side of each failed goal an arrowed line points to the goal to which standard backtracking returns. On the left-hand side of the trace another arrowed line points from the first failed goal to the point further down in the trace that

would correspond to an execution with selective backtracking. The segment of trace jumped over corresponds exactly to the useless computation of standard backtracking, as compared with the selective version.



6.2 Relational database example

This example deals with a database of students who take courses, professors who teach courses, and courses held on certain weekdays and rooms. Our example query is

"Is there a student such that a professor teaches
him two different courses in the same room?"

```
query(S,P)←student(S,C1),      1
           course(C1,D1,R),    2
           professor(P,C1),    3
           student(S,C2),      4
           course(C2,D2,R),    5
           professor(P,C2),    6
           C1≠C2               7
```

The example database follows.

```
student(robert,prolog)←

student(john,music)←
student(john,prolog)←
student(john,surf)←

student(mary,science)←
student(mary,art)←
student(mary,physics)←

professor(luis,prolog)←
professor(luis,surf)←

professor(antonio,prolog)←

professor(eureka,music)←
professor(eureka,art)←
professor(eureka,science)←
professor(eureka,physics)←

course(prolog,monday,room1)←
course(prolog,friday,room1)←

course(surf,sunday,beach)←

course(maths,tuesday,room1)←
course(maths,friday,room2)←

course(science,thursday,room1)←
course(science,friday,room2)←

course(art,tuesday,room1)←

course(physics,thursday,room3)←
course(physics,saturday,room2)←
```

The first student and course picked up in goal 1 are robert and prolog, which are again picked up in goal 4. Execution fails at 7, where C1 and C2 are tested to be different. Both C1 and C2 contribute to failure. The binding of C1 (prolog) was obtained at 1, and that of C2 (prolog) at 4. So the set of backtrack goals is (0,1,4), where 0 is the parent goal. Consequently, backtracking jumps to 4 over 6 and 5.

Goal 4 fails since robert takes no more courses. An analysis would now ensue of the failed matches of goal 4. However, in any sized database, that analysis is too expensive. A simple solution to this problem consists in doing no analysis at all, and in taking as modifying goals all the modifying goals associated with the bindings of all the arguments of the goal. Certainly no solutions are lost in doing so, and probably any one constant name in the bindings would contribute with the most ancient modifying goal in some clause, given the diversity of the database.

To continue with the example, at goal 4 backtrack goal 1, corresponding to the binding of robert to S, is re-selected. Since C2 is still free it renders no modifying goal. The set of backtrack goals becomes (0,1).

Backtracking now jumps to 1 over 3 and 2. This time john and music are picked up. Failure occurs again at 7 but, after backtracking is made to 4, another solution is found for C2, namely prolog. However, the professor of music is not a professor of prolog, so 6 fails. Because the bindings of P and C2 have originated at 3 and 4, respectively, the set is updated to (0,3,4).

And so on and so back and forth.

It should be noted that no other ordering of the goals within the query could prevent selective backtracking from being advantageous over standard backtracking.

6.3 Non-attacking chessboard queens example

The problem consists in placing n chess queens on a nxn chessboard such that no two queens attack one another. The program is general, only the input changes according to the number of queens. For four queens it is

```
←perm(4.3.2.1.nil,L),pair(4.3.2.1.nil,L,Q),safe(Q)
```

The position of a queen is a pair of coordinates. 'perm' generates a permutation of row positions for the four queens, thereby ensuring that no two queens occupy the same row. 'pair' pairs each element of the list L of row positions with a different column, thereby ensuring that no two queens occupy the same column. 'safe' inspects whether any two queens are on the same diagonal.

The program follows.

```
perm(nil,nil)←
perm(X.Y,U.V)←delete(U,X.Y,W),perm(W,V)

delete(X,X.Y,Y)←
delete(U,X.Y,X.V)←delete(U,Y,V)

pair(nil,nil,nil)←
pair(X.Y,U.V,p(X,U).W)←pair(Y,V,W)

safe(nil)←
safe(Q.R)←check(Q,R),safe(R)

check(Q,nil)←
check(Q,R.S)←not_on_diagonal(Q,R),check(Q,S)

not_on_diagonal(p(C1,R1),p(C2,R2))←minus(C1,C2,C),
                                   minus(R1,R2,R),
                                   C≠R,
                                   minus(R2,R1,NR),
                                   C≠NR
```

'safe' uses 'check' to check each pair against all the following pairs in the list. This is accomplished by subtracting corresponding coordinates with the predicate 'minus'.

Backtracking occurs when two queens are on the same diagonal. Standard backtracking would generate the next permutation, even though it may not alter the coordinates of the conflicting queens. Selective backtracking will return to the point in the generation of the permutation where one of the conflicting queens is assigned a different row if possible.

The first permutation being 4.3.2.1.nil, the first two pairs, p(4,4) and p(3,3), are the first to conflict. Standard backtracking would then permute the rows of the queens on columns 1 and 2 to no avail. Selective backtracking begins by detecting that the conflicting constants binding C and R (1 and 1) at goal C≠R were obtained at minus(C1,C2,C) and minus(R1,R2,R), and selects them as backtrack goals. Since there are no alternatives for these calls, they fail. Which are the modifying goals for the current values of C1,C2,R1 and R2? These values were transmitted through a chain of ancestors up to the first call to 'safe', but such dependencies are implicit. The two first calls to 'pair' originated bindings responsible for the transmission of the values, so they are modifying goals. Also, the presence of the values for R1 and R2 in those two first calls to 'pair' depends on the two first calls to 'delete', where they were transmitted from the second to the first argument of the first clause head for 'delete'; these must therefore be also modifying goals. The transmission of the values from perm(4.3.2.1.nil,L) down to the two first calls to 'delete' and then up again is again implicitly accounted for.

To sum up, the successive backtrack goals generated upon failure of C≠R are: first, minus(4,3,C), minus(4,3,R) and the parent not_on_diagonal(p(4,4),p(3,3)); next, on failure of minus(4,3,R), the second call to 'pair', pair(3.2.1.nil,3.2.1.nil,W), and the second call to 'delete', delete(U,3.2.1.nil,W); next, on failure of the call minus(4,3,C), the first call to 'pair', pair(4.3.2.1.nil,4.3.2.1.nil,Q), and the first call to 'delete', delete(U,4.3.2.1.nil,W).

Finally, everytime one of these backtrack goals fails because there are no more clauses for it, its parent goal is included as a backtrack goal, and an analysis takes place to find reasons for any failure of the goal in matching the clauses. Thus, when 'not_on_diagonal' fails, there being no failure in entering its single clause, only its parent, 'check', is selected. But 'check' also fails, so an analysis takes place to examine why its first clause failed to match the goal. The reason is the conflict between the principal functor '.' in the binding of R and 'nil'. A modifying goal for that conflict is the goal in whose match that functor was obtained. It turns out to be the second call to 'pair', already retained as a backtrack goal. The fact is that all constant components of p(4,4).W were obtained in the match of the second call of 'pair'. Similarly, analysis of the failure of 'safe', ie. the conflict between the binding of Q and 'nil', singles out the first call of 'pair' as the culprit, already retained as a backtrack goal.

References

[1] Bruynooghe, M.
Intelligent backtracking for Horn clause logic programs
Colloquium on Mathematical Logic in Programming
Salgotarjan, Hungary 1978

[2] Coelho, H. ; Cotta, J.C. ; Pereira, L.M.
How to solve it with Prolog
Laboratório Nacional de Engenharia Civil, Lisboa 1979

[3] Kowalski, R.A.
Predicate logic as a programming language
IFIP 74, pp 569-574, North-Holland Publ. Co. 1974

[4] Kowalski, R.A.
Logic for problem solving
North-Holland Publ. Co. 1980

[5] Loveland, D.
Automated theorem proving: a logical basis
North-Holland Publ. Co. 1978

[6] Pereira, L.M. ; Pereira, F.C.N. ; Warren, D.H.D.
User's guide to DECsystem-10 Prolog
Laboratório Nacional de Engenharia Civil, Lisboa 1978

[7] Pereira, L.M.
Backtracking intelligently in AND/OR trees
Departamento de Informática
Universidade Nova de Lisboa, Lisboa 1979

(8) Pereira, L.M. ; Porto, A.
Intelligent backtracking and sidetracking
in Horn clause programs - the theory
Departamento de Informática
Universidade Nova de Lisboa, Lisboa 1979

(9) Pereira, L.M. ; Porto, A.
An interpreter of logic programs using selective backtracking
Paper submitted to the Workshop on Logic Programming, organized
by the von Neumann Computer Society, Budapest July 1980

(10) Roussel, P.
Prolog: manuel de référence et d' utilisation
Groupe d' Intelligence Artificielle,
Université d' Aix-Marseille II 1975

(11) Warren, D.H.D.
Implementing Prolog, Parts I and II
Department of Artificial Intelligence
Edinburgh University 1977

(12) Warren, D.H.D. ; Pereira, L.M. ; Pereira, F.C.N.
Prolog, the language and its implementation compared with Lisp
ACM Symposium on Artificial Intelligence and Programming Languages,
Sigart Newsletter no. 64, and Sigplan Notices vol. 12, no. 8, August 1977