Published in Reflection 99, LNCS N° 1616

Metaprogramming domain specific metaprograms

Tristan Cazenave

Laboratoire d'Intelligence Artificielle,
Département Informatique, Université Paris 8,
2 rue de la Liberté,
93526 Saint Denis, France.
cazenave@ai.univ-paris8.fr

Abstract. When a metaprogram automatically creates rules, some created rules are useless because they can never apply. Some metarules, that we call impossibility metarules, are used to remove useless rules. Some of these metarules are general and apply to any generated program. Some are domain specific metarules. In this paper, we show how dynamic metaprogramming can be used to create domain specific impossibility metarules. Applying metaprogramming to impossibility metaprogramming avoids writing specific metaprogram for each domain metaprogramming is applied to. Our meta-metaprograms have been used to write metaprograms that write search rules for different games and planning domains. They write programs that write selective and efficient search programs.

1 Introduction

Knowledge about the moves to try enables to select a small number of moves from a possibly large set of possible moves. It is very important in complex games and planning domains where search trees have a large branching factor. Knowing the moves to try drastically cuts the search trees. Metaprogramming can be used to automatically create the knowledge about interesting and forced moves, only given the rules about the direct effects of the moves [4],[5]. Impossibility metaprograms enable to remove useless rules from the set of unfolded rules. These metaprograms can themselves be written by metametaprograms. From a more general point of view, metaknowledge itself can be very useful for a wide range of applications [23], and one of its fascinating characteristic is that it can be applied to itself to improve itself. We try to experimentally evaluate the benefits one can get from this special property.

The second section describes metaprogramming and especially metaprogramming in games. The third section uncovers how metametaprograms can be used to write impossibility metaprograms. The fourth section gives experimental results.

Published in Reflection 99, LNCS N° 1616

2 Metaprogramming in games and planning domains

Our metaprograms write programs that enable to safely cut search trees, therefore enabling large speedups of search programs. In our applications to games, metarules are used to create theorems that tell the interesting moves to try to achieve a tactical goal (at OR nodes). They are also used to create rules that find the complete set of forced moves that prevent the opponent to achieve a tactical goal (at AND nodes). Metaprogramming in logic has already attracted some interest [11],[2],[9]. More specifically, specialization of logic program by fold/unfold transformations can be traced back to [26], it has been well defined and related to Partial Evaluation in [15], and successfully applied to different domains [9]. The parallel between Partial Evaluation and Explanation-Based Learning [21],[18],[6],[13],[24] is now well-known [28],[7]. As Pitrat [23] points it, the ability for programs to reason on the rules of a game so as to extract useful knowledge to play is a problem essential for the future of AI. It is a step toward the realization of efficient general problem solvers.

In our system, two kinds of metarules are particularly important : impossibility metarules and monovaluation metarules. Other metarules such as metarules removing useless conditions or ordering metarules are used to speed-up the generated programs.

Impossibility metarules find which rules can never be applied because of some properties of the game, or because of more general impossibilities. An example of a metarule about a general impossibility is the following one :

```
impossible(ListAtoms):-
        member(N=\=N1,ListAtoms),var(N),var(N1),N==N1.
```

This metarule tells that if a rule created by the system contains the condition `'N=\=N1'` and the metavariables N and N1 contain the same variable, then the condition can never be fulfilled. So the rule can never apply because it contains a statement impossible to verify. These metarule is particularly simple, but this is the kind of general knowledge a metasystem must have to be able to reason about rules and to create rules given the definition of a game.

Some of the impossibility metarules are more domain specific. For example the following rule is specific to the game of Go :

```
impossible(ListAtoms):-
        member(color_string(B,C),ListAtoms),C==empty.
```

It tells that the color of a string can never be the color `'empty'`.

The other important metarules in our metaprogramming system are the monovaluation metarules. They apply when two variables in the same rules always share the same value. Monovaluation metarules unify such variables. They enable to simplify the rules and to detect more impossible rules. An example of a monovaluation metarule is :

```
monovaluation(ListAtoms):-
          member(color_string(B,C),ListAtoms),
          member(color_string(B1,C1),ListAtoms),
          B==B1,C\==C1,C=C1.
```

It tells that a string can only have one color. So if the system finds two conditions `'color_string'` in a rule such that the variables contained in the metavariables `'B'` and `'B1'` are equals, and if the corresponding color variables contained in the metavariables `'C'` and `'C1'` are not the same, it unifies `C` and `C1` because they always contain the same value.

Impossible and monovaluation metarules are vital rules of our metaprogramming system. They enable to reduce significantly the number of rules created by the system, eliminating many useless rules. For example, we tested the unfolding of six rules with and without these metarules. Without the metarules, the system created 166 391 568 rules by regressing the 6 rules on only one move. Using basic monovaluation and impossible metarules shows that only 106 rules where valid and different from each other.

The experiments described here use the goal of taking enemy stones to create rules for the game of Go. The game of Go is known to be the most difficult game to program [27],[1],[25]. Experiments in solving problems for the hardest game benefit to other games, and to other planning domains, especially if these experiments use general and widely applicable methods as it is the case for our metaprogramming methods.

3 Programs that write programs that write programs

Works on writing programs that write programs that write programs often refer to the third Futamura projection. Their goal is to speed up programs that write programs using self-application of Partial Evaluation. Self-applicable partial evaluators such as Goëdel [11] usually use a ground representation to enable self-application (a ground representation consists in representing variables in the programs by numbers, a parallel can be made with the numbering technique used by the mathematician Kurt Gödel to prove his famous theorem [10]). Our choice is rather to use general non-ground metaprograms that find domain specific metaknowledge to write the programs that write programs. On the contrary of fold/unfold/generalization and other program transformation techniques [20], our system only uses unfolding, and simple metaprograms can be written to decide when to stop unfolding: typically when generated rules have more condition than a pre-defined threshold.

Domain specific metarules in games and planning domains can be divided in different categories. We will focus on the 'board topology metarules' category in this section. Other categories that are often used are 'move metarules' or 'object metarules' for example.

The essential property of a game board is that it never changes whatever the moves are. The set of facts describing the board is always the same. Moreover it is a complete set: no facts can be added or removed.

In this paper, we will use a fixed grid to give examples of metaprogram generation. Grids are used in planning domains, for example for a robot to plan a path through a building, and in games such as Go or Go-Moku.

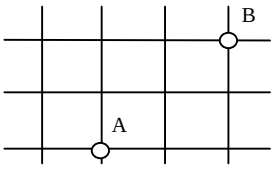

Fig. 1. A robot at point A has to choose a path on the grid to go to point B

The grid task consists in finding a path of length four between point A and point B in the figure 1. This task is easier to understand than the game of Go which is our principal application. A Go board is also a grid, and all the mechanisms described in this paper also apply to the rules generated by our metaprogram for the game of Go.

3.1 Metaprogramming impossibility metarules

The figure 2 gives all the points that are at distance three of point A. After each new instanciation of a variable containing a point, the rule verifies that the instanciated point is different from any previously instanciated one.

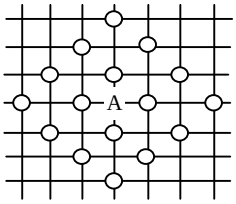

Fig. 2. All the points at a distance three of point A are marked.

The rule that find all these points is generated as follows:

```
distance(X,W,3):-
         connected(X,Y),connected(Y,Z),connected(Z,W).
```

This rule is generated unfolding the goal `'distance(X,W,3)'` defined below:

```
distance(X,X,0).
distance(X,W,N1):-
         N is N1-1,distance(X,Z,N),connected(Z,W).
```

On a grid, all the points that are at a distance two of X are not at a distance three. Moreover, when unfolding definitions in more complex domains, it often happens that two variables are unified, there is only one variable left after unfolding. Unfolding can lead to generate rules similar to this one:

```
distance(X,X,3):-
          connected(X,Y),connected(Y,Z),connected(Z,X).
```

This rule has been correctly generated by a correct metaprogram on a correct domain theory, but it is a rule that will never be applied, because no point on a grid is at a distance three of itself.

We can detect that this rule cannot be fired because we have access to the complete set of facts representing the topology of the board in this domain. In the generated rules, we only select the conditions that are related to the topology of the board (the connected predicates and the test that are done on the variables representing intersections of the grid). Then we fire this set of conditions on the complete set of facts. If the set of conditions never matches, we are confident that the system has generated an impossible set of conditions. In order to find the minimal set of impossible conditions, the system tries to remove the conditions one by one until each removed condition leads to a possible set of conditions. We now have a minimal set of conditions representing a subset of the initial rule that is impossible to match. In our simple example of the distance three goal, it is composed of the three conditions of the rule.

Once this subset is created, it is used to generate a new impossibility metarule. Impossibility metarules match generated rules to find subsets of impossible conditions. If an impossibility metarule succeeds, the generated rule is removed. The impossibility metarule generated in our example is:

```
impossible(ListAtoms):-
          member(connected(X,Y),ListAtoms),
          var(X),var(Y),X\==Y,
          member(connected(A,Z),ListAtoms),
          A==Y,var(A),var(Z),A\==Z,
          member(connected(B,C),ListAtoms),
          B==Z,var(B),var(C),B\==C,C==X.
```

The generation of impossibility metarules is useful in almost all the planning domains we have studied. Here is another example of its usefulness for the game of Abalone.

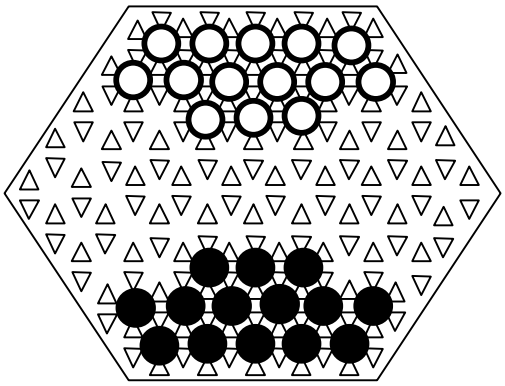

Fig. 3. An abalone board at the beginning of the game

The figure 3 represents an Abalone board, on this board, each position has six neighbors instead of four for the grid board. Each neighbor is associated to one of the six directions represented by numbers ranging from 1 to 6.

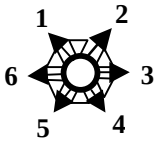

Fig. 4. The directions a stone can be moved to are marked, ranging from one to six.

In the rules of the game and therefore in the rules generated by our metaprogram, the connected predicates contain a slot for the direction. Here is an example of an impossible set of conditions in the game of Abalone :

```
connected(X,Y,Direction),connected(Y,X,Direction),
```

The corresponding impossibility metarule for the game of Abalone is:

```
impossible(ListAtoms):-
        member(connected(X,Y,D),ListAtoms),
        member(connected(A,B,C),ListAtoms),
        A==Y,B==X,C==D.
```

It is also generated using our metametaprogram that generate impossibility metarules.

3.2 Metaprogramming simplifying metarules

The system sometimes generates some rules that contain useless conditions. We give below a rule that finds a path of length four to go from one point of a grid to another one without going twice through the same point. After each new instancia-

tion of a variable in the conditions, the rule verifies that the instanciated point is different from any previously instanciated one.

After each condition in the rule, we give the number of times the condition has been verified when matching the rule once on a set of facts.

```
distance(X,D,4):-
        connected(X,Y),                  4
        X=\=Y,                           4
        connected(Y,Z),                 16
        Z=\=X,                          12
        Z=\=Y,                          12
        connected(Z,W),                 48
        W=\=X,                          48
        W=\=Y,                          36
        W=\=Z,                          36
        connected(W,D).                144
```

However, in some cases, it is useless to verify that some points are different due to the topology of the grid. For example, two connected points are always different. We can use a metarule that tells to remove the condition `'X=\=Y'` if the condition `'connected(X,Y)'` is also present in the rule:

```
useless(X=\=Y,ListAtoms):-
        member(connected(X,Y),ListAtoms),
        member(A=\=B,ListAtoms),A==X,B==Y.
```

Another metarule, given below, removes the condition `'X=\=Y'` when there is a path of length three between the two points contained in `X` and `Y`, this is a consequence of the figure 2 that shows all the points that are at a three step path from point A: A is not at a three step path of itself.

```
useless(X=\=Z,ListAtoms):-
        member(connected(X,Y),ListAtoms),
        var(X),var(Y),X\==Y,
        member(connected(Y,Z),ListAtoms),
        var(Y),var(Z),Y\==Z,
        member(connected(Z,A),ListAtoms),
        var(Z),var(A),Z\==Y,A==X.
```

The initial rule makes 361 instanciations and tests. After firing the metarule of deletion on the initial rule, we obtain the rule below that only makes 261 instanciations or tests with the same results.

```
distance(X,D,4):-
        connected(X,Y),                  4
        connected(Y,Z),                 16
        Z=\=X,                          12
        connected(Z,W),                 48
```

```
W=\=Y,                          36
connected(W,D).                144
```

The simplifying metarules described here can also be generated using a complete set of facts representing the board topology.

For example, if our system analyzes a rule containing the conditions:

```
connected(X,Y),X=\=Y,
```

in this order. It observes that the condition `'X=\=Y'` is always fulfilled. So it tries to remove all the conditions of the rule one by one, provided the condition `'X=\=Y'` is always fulfilled when matching the set of remaining conditions. At the end of this process, the final set of conditions only contains the two above conditions, and the condition `'X=\=Y'` is always fulfilled for all the complete set of facts in the working memory. As the set of fact representing the topology of the board is complete, it can generate a new simplifying metarule that will apply to all the generated rules of this domain. This simplifying metarule is the one given above.

This method works in all planning domains where a complete set of facts can be isolated, such as the topology of the board, for games where the topology cannot be changed by the moves.

3.3 Metaprogramming ordering metarules

Related Work

P. Laird [14] uses statistics on some runs of a program to reorder and to unfold clauses of this program. T. Ishida [12] also dynamically uses some simple heuristics to find a good ordering of conditions for a production system. Our approach is somewhat different, it takes examples of working memories to create metarules that will be used to reorder the clauses. What we do, is automatically creating a metaprogram that is used to reorder the clauses, and not dynamically reordering conditions of the rules. One advantage is that we can create this metaprogram independently. Moreover, once the metaprogram is created, running it to reorder learned rules is faster than dynamically optimizing the learned rules. This feature is important for systems that use a large number of generated rules. The creation of the metaprogram is also fast.

We rely on the assumption that domain-dependent information can enhance problem solving [16]. This assumption is given experimental evidence on constraint satisfaction problems by S. Minton [17]. On the contrary of Minton, we do not specialize heuristics on specific problems instances, we rather create metaprograms according to specific distributions of working memories.

Reordering conditions

Reordering conditions is very important for the performance of generated rules. The two following rules are simple examples that show the importance of a good order

of conditions. The two rules give the same results but do not have the same efficacy when X is known and Y unknown:

```
sisterinlaw(X,Y):-brother(X,X1),married(X1,Y),woman(Y).
sisterinlaw(X,Y):-woman(Y),brother(X,X1),married(X1,Y).
```

Reordering based only on the number of free variables in a condition does not work for the example above. In the constraint literature, constraints are reordered according to two heuristics concerning the variables to bind [17] : the range of values of the variables and the number of other variables it is linked to. These heuristics dynamically choose the order of constraints. But to do so, they have to keep the number of possible bindings for each variable, and to lose time when dynamically choosing the variable. It is justified in the domain of constraints solving because the range of value of a variable, affects a lot efficiency, and can change a lot from one problem to another. It is not justified in some other domains where the range of value a variable can take is more stable. We have chosen to order conditions, and thus variables, statically by reordering once for all and not dynamically at each match because it saves more time in the domains in which we have tested our approach.

Reordering optimally the conditions in a given rule is an NP-complete problem. To reorder conditions in our generated rules, we use a simple and efficient algorithm. It is based on the estimated number of following nodes the firing of a condition will create in the semi-unification tree. Here are two metarules used to reorder conditions of generated rules in the game of Go:

```
branching(ListAtoms,ListBindVariables,
          connected(X,Y),3.76):-
          member(connected(X,Y),ListAtoms),
          member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).

branching(ListAtoms,ListBindVariables,
          elementstring(X,Y),94.8):-
          member(elementstring(X,Y),ListAtoms),
          non_member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).
```

A metarule evaluates the branching factor of a condition based on the estimated mean number of facts matching the condition in the working memory. Metarules are fired each time the system has to give a branching estimation for all the conditions left to be ordered. When reordering a rule containing N conditions, the metarule will be fired N times: the first time to choose the condition to put at first in the rule, and at time number T to choose the condition to put in the T^{th} place. In the first reordering metarule above, the variable X is already present in some of the conditions preceding the condition to be chosen. The variable Y is not present in the preceding conditions. The condition `'connected(X,Y)'` is therefore estimated to have a branching factor of 3.76 (this is the mean number of neighbor intersections of an

intersection on a 19*19 grid, this number can vary from 2 to 4), this is the mean number of bindings of Y.

The branching factors of all the conditions to reorder are compared and the condition with the lowest branching factor is chosen. The algorithm is very efficient, it orders rules better than humans do and it runs fast even for rules containing more than 200 conditions.

Generating ordering metarules

For each predicate in the domain theory that has an arity less or equal than three. Each variable of the predicate free or not, leading to 2^3=8 possibilities for the three variables. So, for each predicate, we create between 1 and 8 metarules.

For predicates of arity greater than three, we only create the metarules that corresponds to the bindings of all but one of the variables of the predicate.

All the metarules are tested on some working memories. This enables to conclude on the priority to give to the metarule. The priority is the mean number of bindings the condition will create. The lower the priority, the sooner the condition is to be matched. When all the variables of a condition are instanciated, it is a test and it has a priority between zero and one, whereas predicates containing free variables have, most of the time, a priority greater than one.

4 Results

This section gives the results and the analysis of some experiments in generating metaprograms. We used a Pentium 133 with SWI-Prolog for testing.

4.1 Metaprogramming impossibility metarules

In the figure 5, the horizontal axis represents the number of rules unfolded by our metaprogram on one move. This experiment was realized using the game of Go domain theory associated to the subgoal of taking stones of the opponent. There are six unfolded rules, it means that six rules concluding on a won subgoal of taking stones where randomly chosen out of the total number of such rules created by our system. Each of these six rules has been unfolded using the rules of the game of Go, without the cuts of impossibility and monovaluation metarules. All of the six rules to unfold, match Go boards where the friend color can take the opponent string in less than two moves, whatever the opponent plays. The goal of the unfolding was to find all the rules that find a move that lead to match one of the six rules concluding on a won state. The vertical axis of the figure 5 represents the cumulated number of rules that have been created for each of the six rules. We did not match the impossible and monovaluation metarules on the resulting rules because it would have been to time consuming.

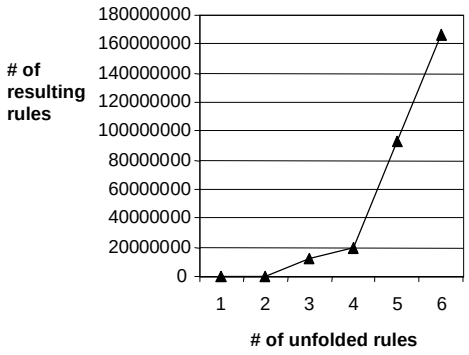

Fig. 5. Number of rules generated when unfolding six simple rules without impossibility and monovaluation metarules.

Instead of unfolding all the rules one move ahead and then destroying useless rules, we matched the monovaluation and impossibility metarules after each unfolding step (an unfolding step is the replacement of a predicate by one of its definitions). Each unfolding step is considered as a node in the unfolding tree.

Fig. 6. Number of rules generated when unfolding six simple rules with impossibility and monovaluation metarules

This resulted in a very significant improvement of the specialization program, it was much faster and completely unfolding the six rules only gave 106 resulting rules concluding on winning moves to take stones 3 moves in advance. The results are shown in the figure 6, and can be compared with the results of the figure 5. It is important to see that among all the resulting rules of the figure 5, only the 106 resulting rules of the figure 6 are valid and different from each other.

This experiment also stresses the importance of the impossibility metarules. Without them, unfolding a goal on a domain theory is not practically feasible. Therefore impossibility metarules are necessary for such programs, and automatically generating them is a step further in the automatisation of planning programs development.

4.2 Metaprogramming ordering metarules

When no metarule concludes on the priority of the conditions left to be ordered, simple reordering heuristics are used. For example, the condition containing the less variables is chosen.

Following are two equivalent rules. The first one is ordered without metarules, and the second one is ordered using the learned metarules :

```
threattoconnect(C,B,B1,I):-
      colorintersection(I1,empty),      55
      connected(I,I1),                 208
      connected(I1,I2),                796
      I=\=I2,                          588
      liberty(I2,B1),                  306
      colorblock(B1,C),                140
      liberty(I,B),                     84
      colorblock(B,C),                  36
      color(C).                         36
                                       2249

threattoconnect(C,B,B1,I):-              1
      color(C),                          2
      colorblock(B,C),                  14
      liberty(I,B),                     68
      connected(I,I1),                 240
      colorintersection(I1,empty),      96
      connected(I1,I2),                350
      I=\=I2,                          254
      liberty(I2,B1),                   84
      colorblock(B1,C).                 36
                                       1145
```

Each condition is followed by the number of time it has been accessed during the matching of the rule. In this example, when choosing the first condition, a classic order gives `'colorintersection(I,empty)'` in the first rule. In the second rule, the two following metarules where matched among others to assign priorities to conditions :

```
branching(ListAtoms,ListBindVariables,
          colorintersection(I,C),240.8):-
          member(colorintersection(I,C),ListAtoms),
          C==empty,
          non_member_term(I,ListBindVariables).

branching(ListAtoms,ListBindVariables,color(C),2):-
          member(color(C),ListAtoms),
          non_member_term(C,ListBindVariables).
```

Therefore, the condition `'color(C)'` has been chosen because it has the lowest branching factor.

The two rules given in the example are simple rules. Speedups are more important with rules containing more conditions.

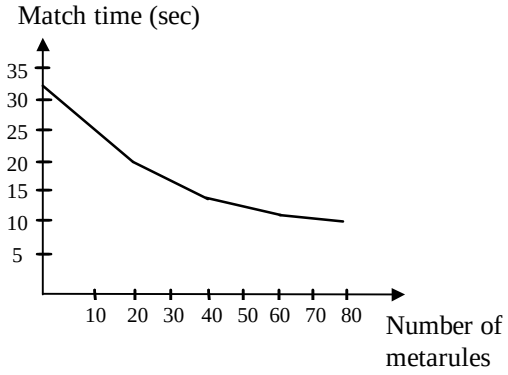

Fig. 7. The match time of generated rules decreases when more ordering metarules are generated and used.

The figure 7 gives the evolution of the matching time of a set of generated rules with the number of metarules generated. The evolution is computed on a test set of 50 problems. Problems in the test set are different from the problems used to generate the metarules.

5 Conclusion

Metaprogramming games and planning domains is considered as an interesting challenge for AI [19],[23]. Moreover it has advantages over traditional approaches: metaprograms automatically create the rules that otherwise take a lot of time to create, and the results of the search trees developed using the generated programs are more reliable than the results of the search trees developed using traditional heuristic and hand-coded rules.

The Go program that uses the rules resulting of the metaprogramming has good results in international competitions (6 out of 40 in 1997 FOST cup [8], 6 out of 17 in 1998 world computer Go championship). The metaprogramming methods presented here can be applied in many games and in other domains than games. They have been applied to other games like Abalone and Go-Moku, and to planning problems [3]. Using metaprogramming this way is particularly suited to automatically create complex, efficient and reliable programs in domains that are complex enough to require a lot of knowledge to cut search trees.

However metaprogramming large programs can be itself time consuming. We have proposed and evaluated methods to apply metaprogramming to itself so as to

make it more efficient. These methods gave successful results. Moreover they tend to give even better results when generated programs become more complex.

6 References

1. Allis, L. V.: Searching for Solutions in Games an Artificial Intelligence. Ph.D. diss., Vrije Universitat Amsterdam, Maastricht 1994.
2. Barklund J. : Metaprogramming in Logic. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.
3. Cazenave, T.: Système d'Apprentissage par Auto-Observation. Application au Jeu de Go. Ph.D. diss., Université Paris 6, 1996.
4. Cazenave T.: Metaprogramming Forced Moves. Proceedings ECAI98, Brigthon, 1998.
5. Cazenave T.: Controlled Partial Deduction of Declarative Logic Programs. ACM Computing Surveys, Special issue on Partial Evaluation, 1998.
6. Dejong, G. and Mooney, R.: Explanation Based Learning : an alternative view. Machine Learning 1 (2), 1986.
7. Etzioni, O.: A structural theory of explanation-based learning. Artificial Intelligence 60 (1), pp. 93-139, 1993.
8. Fotland D. and Yoshikawa A.: The 3rd fost-cup world-open computer-go championship. ICCA Journal 20 (4):276-278, 1997.
9. Gallagher J.: Specialization of Logic Programs. Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.
10. Gödel K.: 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I', Monatsh. Math. Phys. 38, 173-98, 1931.
11. Hill P. M. and Lloyd J. W.: The Gödel Programming Language. MIT Press, Cambridge, Mass., 1994.
12. Ishida T.: Optimizing Rules in Production System Programs, AAAI 1988, pp 699-704, 1988.
13. Laird, J.; Rosenbloom, P. and Newell A. Chunking in SOAR : An Anatomy of a General Learning Mechanism. Machine Learning 1 (1), 1986.
14. Laird P.: Dynamic Optimization. ICML-92, pp. 263-272, 1992.
15. Lloyd J. W. and Shepherdson J. C.: Partial Evaluation in Logic Programming. J. Logic Programming, 11 :217-242., 1991.
16. Minton S.: Is There Any Need for Domain-Dependent Control Information : A Reply. AAAI-96, 1990.
17. S. Minton. Automatically Configuring Constraints Satisfaction Programs : A Case Study. Constraints, Volume 1, Number 1, 1996.
18. Mitchell, T. M.; Keller, R. M. and Kedar-Kabelli S. T.: Explanation-based Generalization : A unifying view. Machine Learning 1 (1), 1986.
19. Pell B.: A Strategic Metagame Player for General Chess-Like Games. Proceedings of AAAI'94, pp. 1378-1385, 1994. ISBN 0-262-61102-3.
20. Pettorossi, A. and Proietti, M.: A Comparative Revisitation of Some Program Transformation Techniques. Partial Evaluation, International Seminar, Dagstuhl Castle, Germany LNCS 1110, pp. 355-385, Springer 1996.

21. Pitrat J.: Realization of a Program Learning to Find Combinations at Chess. Computer Oriented Learning Processes, J. C. Simon editor. NATO Advanced Study Institutes Series. Series E: Applied Science - N° 14. Noordhoff, Leyden, 1976.
22. Pitrat, J.: Métaconnaissance - Futur de l'Intelligence Artificielle. Hermès, Paris, 1990.
23. Pitrat, J.: Games: The Next Challenge. ICCA journal, vol. 21, No. 3, September 1998, pp.147-156, 1998.
24. Ram, A. and Leake, D.: Goal-Driven Learning. Cambridge, MA, MIT Press/Bradford Books, 1995.
25. Selman, B.; Brooks, R. A.; Dean, T.; Horvitz, E.; Mitchell, T. M.; Nilsson, N. J.: Challenge Problems for Artificial Intelligence. In Proceedings AAAI-96, 1340-1345, 1996.
26. Tamaki H. and Sato T.: Unfold/Fold Transformations of Logic Programs. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.
27. Van den Herik, H. J.; Allis, L. V.; Herschberg, I. S.: Which Games Will Survive ? Heuristic Programming in Artificial Intelligence 2, the Second Computer Olympiad (eds. D. N. L. Levy and D. F. Beal), pp. 232-243. Ellis Horwood. ISBN 0-13-382615-5. 1991.
28. Van Harmelen F. and Bundy A.: Explanation based generalisation = partial evaluation. Artificial Intelligence 36:401-412, 1988.