Controlled Partial Evaluation of Declarative Logic Programs. Synthesis of an efficient Tactical Theorem Prover for the Game of Go.

Tristan Cazenave
LIP6, Université Pierre et Marie Curie

1. INTRODUCTION

Partial evaluation of logic programs [Lloyd and Shepherdson 1991], also known as partial deduction, is a powerful tool for program specialization and has been successfully applied to the derivation of very efficient specialized programs [Gallagher 1993]. Logic Programming provides a nice and convenient way to represent knowledge. An important goal of Logic Programming is declarativity, which involves stating *what* is to be computed, but not necessarily *how* it is to be computed. In the terminology of Kowalski's equation *algorithm = logic + control*, it involves stating the logic of an algorithm, but not necessarily the control. Giving only the logic of an algorithm is very convenient and enables to give easily a lot of knowledge to a program, however it is very inefficient and often leads to a combinatorial explosion in the application of the algorithm. Introspect [Cazenave 1996] is a system that transforms a concise but inefficient declarative logic program into an efficient one by unfolding the goals specified by the programmer. It uses domain specific metaprograms [Barklund 1994] [Hill and Lloyd 1994] to control unfolding and to rewrite object programs. Metaprograms are also logic programs that use some built-in metapredicates.

Introspect has been used to partially evaluate some goals of the game of Go, the most complex two person complete information game. It uses the rules of the game represented declaratively in first order predicate logic and some metaprograms to unfold the goals. The partially evaluated program has been compiled into a 1 000 000 lines C++ program that develops tactical proof trees. This program competed

Address: LIP6, Université Pierre et Marie Curie Case 169, 4 place Jussieu, 75252 PARIS cedex 05, France. e-mail: Tristan.Cazenave@lip6.fr

in the international computer Go tournament held during IJCAI97 together with 40 other participants [Fotland and Yoshikawa 1997]. It finished as the best program based on academic research, playing better that the other programs directly written by Artificial Intelligence researchers and Go professionals. It has outperformed programs that have required more than 10 person*years of professional game programmers.

Introspect has also been applied in other domains and, in these domains too, it has written C++ programs that give better results than programs written by professional C++ programmers.

Lloyd and Shepherdson [Lloyd and Shepherdson 1991] conclude that "the control of partial evaluation is one of the most difficult issues that need to be resolved before partial evaluation can realize its full potential". We have a bottom-up approach to this problem: we write metaprograms that enable a good control of partial evaluation in a complex domain (i.e. the game of Go), then we generalize the control metaprograms by applying Introspect to multiple domains (i.e. pedestrian simulation, other games...). We do not rely on a fixed strategy for choosing clauses: in our programs the clauses may be used in any order, the programs give the same results. That is why we say our logic programs are declarative: the clauses in the programs can be put in any order.

In the following, we give reasons for Go to be a valuable challenge for computer science. We briefly explain how actual Go programs are made and the interest of automatic program generation for the game of Go. Next, we sketch the controlled partial evaluation process of Introspect. At the end, we describe some applications.

2. COMPLEXITY OF GO

Go was developed three to four millennia ago in China; it is the oldest and one of the most popular board game in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. In spite of the simplicity of its rules, playing the game of Go is a very complex task. Robson [Robson 1983] proved that the time complexity of Go generalized to NxN boards is exponential in N. More concretely, van den Herik [van den Herik et al. 1991] and Allis [Allis 1994] define the whole game tree complexity $A = B^L$, where L is the average length of a game and B is the average branching factor. The state-space complexity of a game is defined as the number of legal game positions reachable from the initial position of the game. In Go, $L \approx 150$ and $B \approx 250$ hence the game tree complexity $A \approx 10^{360}$. Go state space complexity, bounded by $3^{361} \approx 10^{172}$, and game tree complexity are far larger than those of any other perfect-information game. Moreover, a position takes time to be evaluated, on the contrary of Chess where the best programs can evaluate a position very fast. This makes Go a challenging programming task.

3. METHODS FOR PROGRAMMING GO

As it is impossible to search the entire tree for the game of Go, the best Go playing programs rely on a knowledge intensive approach. They are generally divided in two modules:

- A tactical module that develops narrow and deep search trees. Each tree is related to the achievement of a goal of the game of Go.

- A strategic module which chooses the move to play according to the results of the tactical module.

A Go expert uses a large number of rules. Go programmers usually try to enter these rules by hand in a Go program. Creating this large number of rules requires a high level of expertise, a lot of time and a long process of trial and error. Moreover, even the people who are expert in Go and in programming find it difficult to design these rules. This phenomenon can be explained by the high level of specialization of these rules: once the expert has acquired them, they become subconscious and it is hard and painful for the expert to explain why he has chosen to consider a move rather than another one. Moreover, even when the work of extracting some rules has been done, it results in thousands of specific expert rules. Thus, it is difficult to describe them in a systematic way.

4. COMPUTER GO CAN BENEFIT FROM PARTIAL EVALUATION

The difficulty of encoding Go knowledge is the consequence of a well known difficulty of expert system development: the knowledge engineering bottleneck. Automatic program generation by Partial Evaluation [Jones et al. 1993] [Consel and Danvy 1993] is a nice way to avoid this bottleneck by replacing the knowledge extraction process with an automated construction of programs. Partial Evaluation enables Go programmers to get rid of the painful expert knowledge acquisition: they only have to define a goal and the rules of the game in predicate logic, partial evaluation automatically writes all the rules that enable to develop search trees to prove the defined goals. Using partial evaluation, programmers only have to define about 70 simple and intuitive rules in first order predicate logic. Without using it, they have to hand-code thousands of complex expert rules (which is currently what all but one Go programmers do). Thus, computer Go is an ideal domain to test the efficacy of Partial Evaluation.

5. CONTROLLED PARTIAL EVALUATION

Partial Evaluation in Introspect is controlled by four metaprograms:

- A metaprogram that detects in a clause the variables that are always equal, and replaces them by a unique variable. For example, two different variables that both contain the number of neighbors of a given intersection are always equal, so they can be merged in a single variable. Introspect always merges variables when it is possible. However when it has created two clauses and when one clause implies the other one, it deletes the more specific one. This metaprogram also performs constant propagation and expression simplification. It contains approximately 50 clauses.

- A metaprogram that removes clauses that can never apply. When partially evaluating a declarative logic program, a large part of the object program is useless because of some properties of the domain. For example in the game of Go, intersections never have more than four neighbors, so the metaprogram removes all the clauses that contain conditions in which one intersection has more than four different neighbors because they will never apply. This metaprogram contains approximately 40 clauses.

- A metaprogram that indicates when to stop unfolding. The definition of a goal N moves ahead is given using information on the goal N-1 moves ahead, therefore it is recursive. Blindly unfolding a goal defined recursively does not terminate. Therefore a simple metaprogram is used to stop unfolding, the criteria for stopping are based on the complexity and on the number of unfolded clauses. The evaluation of the complexity of a clause is based on its length and on the number of moves the clause advises (clauses that advise more than five moves at an AND node are discarded).

- A metaprogram that reorders the atoms inside the clauses so as to unify clauses faster. This metaprogram uses statistical information on the presence of facts in the working memory. It orders the clauses with heuristics that combine information on the number of values a variable can take and on the number of other variables a variable is linked to, so as to make the most informative choices first.

When the controlled Partial Evaluation ends, the resulting set of clauses is put into a tree of atoms and compiled to a C++ program by another metaprogram.

6. APPLICATIONS

In the application to the game of Go, 70 simple clauses define the goals and the rules of the game. The controlled Partial Evaluation and the compilation to C++ of these clauses give the Go program. The clauses unfolded by Introspect and compiled to C++ are theorems about the moves to try when developing a proof tree. Out of the 250 possible moves on a Go board, the clauses select between 1 and 5 moves that are proven to be the only ones that are useful to look at. This exponentially decreases the complexity of the search tree. Comparing the unspecialized program to the specialized program is difficult for the game of Go because the unspecialized program often does not terminate (when there is no solution) and is very slow (it tries all the possible moves), whereas the specialized program always terminates (due to the control metaprogram that stops unfolding) and is quite fast. The compilation to C++ enables the specialized program to run 60 times faster than the specialized logic program. The module that compiles logic programs to C++ has to know the range of values of each variable in each predicate. The Go program plays a move in 10 seconds on a Pentium 133 MHz. Before playing a move, it proves about 450 tactical theorems, each theorem requires between 4 and 600 nodes in a search tree to be proved. At each node of each tree, the C++ program written by Introspect is called to find the useful moves to try. The theorems proved are represented using an extension of combinatorial game theory [Conway 1976] to unknown values [Cazenave 1996].

Introspect has also been used to rewrite the decision part of a pedestrian in a commercial urban simulation. Using controlled Partial Evaluation, it has written a C++ program that is 5 to 10 times faster than the original C++ program written by the authors of the simulation. Other successful applications have been designed for the game of Abalone, and for the optimization of the decision making in the management of a firm.

7. CONCLUSION

Controlled Partial Evaluation of declarative logic programs has proved to be useful for rapidly writing efficient and better than professional programs in two different complex domains: pedestrian simulation and the game of Go. A limitation of our current technique is that our control metaprograms work well in games and simulations but are not guaranteed to work in any other domain: an important remaining problem is to characterize domains in which they work well. Another limitation is that the efficiency of partial evaluation is strongly related to the representation used to describe the domain: two semantically equivalent representations can lead to very different specialized programs. Further research is required to identify general principles for finding good representations of domains in the programs to be specialized.

REFERENCES

ALLIS, L. V. 1994. *Searching for Solutions in Games and Artificial Intelligence.* Ph. D. thesis, Vrije Universitat Amsterdam, Maastricht.

BARKLUND, J. 1994. Metaprogramming in logic. Technical Report 80, UPMAIL, University of Uppsala, Sweden.

CAZENAVE, T. 1996. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go.* Ph. D. thesis, Université Pierre et Marie Curie, Paris 6.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on partial evaluation. In *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages* (1993).

CONWAY, J. 1976. *On Numbers and Games.* Academic Press, London/New-York.

FOTLAND, D. AND YOSHIKAWA, A. 1997. The 3rd fost-cup world-open computer-go championship. *ICCA Journal 20*, 4 (December), 276–278.

GALLAGHER, J. 1993. Specialization of logic programs. In D. SCHMIDT Ed., *Proceedings of the ACM SIGPLAN Symposium on PEPM93* (1993). ACM Press.

HILL, P. M. AND LLOYD, J. W. 1994. *The Gödel Programming Language.* MIT Press, Cambridge, Mass.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

ROBSON, J. M. 1983. The complexity of go. In *Proceedings IFIP* (1983), pp. 413–417.

VAN DEN HERIK, H. J., ALLIS, L. V., AND HERSCHBERG, I. S. 1991. Which games will survive? In D. N. L. LEVY AND D. F. BEAL Eds., *Heuristic Programming in Artificial Intelligence 2, the Second Computer Olympiad*, pp. 232–243. Ellis Horwood. ISBN 0-13-382615-5.