In *ACM Computing Surveys*, Vol. 30, No. 3, September 98. © ACM

Optimizing Modular Logic Languages

Michele Bugliesi
Dipartimento di Matematica Pura e Applicata, Università di Padova
Via Belzoni 7, 35131 Padova, Italy, michele@math.unipd.it
and
Anna Ciampolini
DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy, aciampolini@deis.unibo.it
and
Evelina Lamma
DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy, elamma@deis.unibo.it
and
Paola Mello
Dipartimento di Ingegneria, Università di Ferrara
Via Saragat, 44100 Ferrara, Italy, pmello@ing.unife.it

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Transformation*

Additional Key Words and Phrases: Partial Deduction, Abstract Interpretation

1. MODULAR LOGIC PROGRAMMING

Several applications in the areas of Software Engineering and Artificial Intelligence require support for modular programming and incremental knowledge organization.

Modular logic programming has been studied along two orthogonal lines of research (see [Bugliesi et al. 1994] for a survey). In the first approach [O'Keefe 1985; Mancarella and Pedreschi 1988; Gaifman and Shapiro 1989; Bossi et al. 1993; Brogi et al. 1992], modularity is conceived as a *meta-linguistic* concept: logic modules are

viewed as independent sub-programs, interpreted as elements of an algebra, and module composition is rendered in terms of various operators (e.g., union, deletion, closure and combinations thereof) of that algebra.

In the second approach, modular programming is based on extending the syntax of programs with new logical connectives that support abstraction mechanisms reminiscent of those found in traditional programming languages. Most of the proposals that subscribe to this view [Giordano et al. 1988; Giordano et al. 1994; Miller 1989; Monteiro and Porto 1989; Lamma et al. 1992] arise from using *implication goals* of the kind $D \supset G$ in the body of clauses: when D is a set of universally quantified clauses and G is a goal, evaluating the implication goal $D \supset G$ in a given set of clauses $\mathcal{P}$ is interpreted operationally as a request to dynamically "load" the clauses in D and "add" them to $\mathcal{P}$ before attempting G, and then discard them after the derivation for G succeeds or fails.

Modules are introduced in this paradigm by allowing collections of clauses to be referred to explicitly by names, and programs to be structured as collections of modules each one dedicated to answer a specific class of queries. Different scope rules between modules, and different policies for module composition are then accounted for relying on the logical meaning (and the corresponding operational semantics) that is associated with the implication connective. Under the intuitionistic interpretation [Miller 1989], implication goals support a programming paradigm with dynamic scope rules, where every active module has free access to the clauses residing in all of the currently loaded modules. Under the classical interpretation [Giordano et al. 1988], implication goals lead to languages with static scope rules that mimic the semantics of blocks and nested modules in traditional programming languages. Relying on other interpretations, implication goals are employed to render the behavior of inheritance- and object-based systems [McCabe 1992; Bugliesi 1992].

In the rest of this paper, we focus on modular logic programming based on implication goals, as they appear to be are more powerful and general. The interested reader is referred to [Bugliesi et al. 1994] for a detailed analysis of the relationships between this approach to modularity and the algebraic one.

2. OPTIMIZATION OF MODULAR LOGIC PROGRAMMING

Several papers in the literature [Lamma et al. 1992; Jayaraman and Nadathur 1991; Bugliesi and Nardiello 1994] have presented implementations of implication goals based on extensions of the Warren Abstract Machine [Warren 1983]. The performance analyses on the compiled code give a satisfactory measure of the effectiveness of theses proposals; yet, the analyses also indicate that a notable source of run-time overhead resides in the call mechanism for predicates: roughly, the cost of call to a predicate is the cost of a look-up access in the set (or *context*) of currently loaded modules needed to determine the appropriate set of clauses defining a predicate call.

Two approaches have been proposed to overcome this run-time overhead. The first, more effective for languages with static scope rules, is based on the application of partial evaluation (partial deduction in this context [Komorowski 1981]). The second, better suited for languages with dynamic scope rules, is based on a bottom-up abstract interpretation [Cousot and Cousot 1992].

2.1 Partial Deduction

The application of partial deduction in the optimization of modular logic programming was proposed in [Bugliesi et al. 1993] extending the standard definition of partial deduction in logic programming [Lloyd and Shepherdson 1991]. In this extension, the idea is to assume that the evaluation of a goal always occurs in a context, i.e. a set of clauses that may dynamically change at the different stages of the computation. As a consequence, the result of the transformation is not only a function of the goal, but also of the initial context where this goal has to be evaluated. The definition of partial deduction proposed in [Bugliesi et al. 1993] captures this idea, and results into a transformation scheme for modular programs that is itself modular, in that it preserves the structure of the original program. In particular, applying the transformation to a goal in a given initial context results in a new, still modular, program where some of (possibly all) the modules occurring in the initial context are replaced by their specialized versions. Preserving the structure of the programs allows an incremental transformation process to take place, in which modules of a given program can be replaced by their specialized versions without affecting the semantics of the program. Furthermore, it makes the partial deduction scheme fully compatible with the compilation techniques presented in [Lamma et al. 1992; Jayaraman and Nadathur 1991].

Following the approach introduced in [Lloyd and Shepherdson 1991], the soundness and completeness of the transformation are proved under appropriate *closedness* conditions on the transformed program, the goal and the initial context. The conditions depend on the different interpretations of the implication connective and the corresponding scope rules. For languages with static scope rules, these conditions are decidable by a syntactic check on the transformed program and the goal. In particular:

- for block- and module-based systems, they just correspond to the conditions introduced in [Lloyd and Shepherdson 1991] for definite logic programs; stronger, but still statically decidable, conditions must be imposed instead in order to prove the same completeness result for inheritance-based systems;
- weaker results hold for languages with dynamic scope rules, as the soundness and completeness proofs require further conditions to be satisfied by the contexts of evaluation arising during the computation of the transformed program.

2.2 Abstract Interpretation

A complement to partial deduction in optimizing modular logic languages with implication goals has been proposed in [Ciampolini et al. 1995], where the authors describe a data- and control-flow analysis that computes static information on the dynamic evolution of the context during the evaluation of a goal.

The analysis is based on the classical framework of abstract interpretation, consisting of a *concrete* domain, an *abstract* domain, and a couple of functions α and γ, respectively, the *abstraction* and the *concretization* functions, connecting the two domains.

The concrete domain chosen for the analysis is an extended Herbrand base where each ground atom is associated with any possible (ordered) collection of program modules. The abstract domain, instead, associates each predicate (as opposed to

ground atom) with any possible ordered collection of program modules. Relying on the definition of the fixed point semantics for the language under consideration (given in [Brogi et al. 1990] in terms of an immediate consequence operator T_P), the authors define the abstract transformation function, T_P^α, as the composition: $T_P^\alpha = \alpha \circ T_P \circ \gamma$. As usual, the abstract analysis of a progra consists of the iterative application of T_P^α, starting from the bottom element of the abstract domain, until the least fixed point is reached. The computed least fixed point is a set of pairs of the form $\langle c, p \rangle$ where c is a *minimal context* that might lead to a successful computation for an atom A with predicate p. Minimal contexts are then used to statically determine which contexts will certainly lead to failure for A, so as to avoid entering certainly useless computations. When a call to p is raised, if the current set of modules in use is not a superset of any of the minimal contexts of p, then the computation fails immediately. If no minimal context exists, then the call can be replaced by a failure by means of a proper program transformation.

The impact of the analysis is particularly relevant when an *explicit* representation of the set of currently loaded modules is maintained by the underlying implementation, as described in [Ciampolini et al. 1997], through a bit-vector representation of contexts.

3. CONCLUSIONS AND PERSPECTIVES

The optimization of modular logic programming is one of the keys to make this programming paradigm appealing to a wider community and adequate for developing practical applications.

We have described two optimization techniques, based respectively on partial deduction [Bugliesi et al. 1993] and abstract interpretation [Ciampolini et al. 1995]. Although the two techniques are inherently different, and to some extent complementary, they do rely on several common notions and technical solutions. For instance, the computation of the *minimal contexts* in [Ciampolini et al. 1995] is based on essentially same idea as the computation of the *deduction context* maintained during partial deduction in [Bugliesi et al. 1993], the difference being that the latter results from a process of top-down evaluation, while the former is computed by bottom-up iteration of the T_P function. Based on this and other common aspects, a tighter integration between the two techniques should be possible, and desirable to achieve more effective speedups on larger classes of programs.

The integration of the two techniques, as well as the application of similar techniques to other presentations of modular and structured logic programming, based on linear logic [Hodas 1994] are subjects of future research.

REFERENCES

Bossi, A., Bugliesi, M., Gabbrielli, M., Levi, G., and Meo, M. C. 1993. Differential Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages* (1993), pp. 359–370. ACM Press.

Brogi, A., Lamma, E., and Mello, P. 1990. Hypothetical Reasoning in Logic Programming: A Semantics Approach. *Information Processing Letters 36*, 285–291.

Brogi, A., Lamma, E., and Mello, P. 1992. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing 11*, 1, 1–21.

Bugliesi, M. 1992. A Declarative View of Inheritance in Logic Programming. In K. Apt Ed., *Proc. Joint Int. Conference and Symposium on Logic Programming* (1992), pp. 113–

130. The MIT Press.

Bugliesi, M., Lamma, E., and Mello, P. 1993. Partial Deduction for Structured Logic Programming. *Journal of Logic Programming 16*, 89–122.

Bugliesi, M., Lamma, E., and Mello, P. 1994. Modularity in Logic Programming. *Journal of Logic Programming 19-20*, 443–502.

Bugliesi, M. and Nardiello, G. 1994. SelfLog: Language and Implementation. In G. Succi and E. Tick Eds., *Implementations of Logic Programming Systems* (1994), pp. 1–15. Kluwer.

Ciampolini, A., Lamma, E., and Mello, P. 1995. Improving the efficiency of dynamic modular logic languages. *Information Processing Letters 58*, 163–170.

Ciampolini, A., Lamma, E., and Mello, P. 1997. An optimized implementation of a dynamic modular logic language. *Software, Concepts and Tools 17*, 148–162.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming 13*, 103–180.

Gaifman, H. and Shapiro, E. 1989. Fully abstract compositional semantics for logic programs. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages* (1989), pp. 134–142. ACM Press.

Giordano, L., Martelli, A., and Rossi, G. 1988. Local definitions with static scope rules in logic languages. In *Proc. FGCS'88 Int. Conference* (1988), pp. 389–396.

Giordano, L., Martelli, A., and Rossi, G. 1994. Structured Prolog: A language for structured logic programming. *Software-Concepts and Tools 15*, 125–145.

Hodas, J. 1994. *Logic Programming in Intuitionistic Linear Logic*. Ph. D. thesis, University of Pennsylvania, Department of Computer and Information Science.

Jayaraman, B. and Nadathur, G. 1991. Implementation techniques for scoping constructs in logic programming. In K. Furukawa Ed., *Proc. 8th Int. Conference on Logic Programming* (1991), pp. 871–886. The MIT Press.

Komorowski, H. J. 1981. A specification of an abstract Prolog machine and its application to Partial Evaluation. Technical Report Dissertation, Linkoping University.

Lamma, E., Mello, P., and Natali, A. 1992. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming 14*, 187–222.

Lloyd, J. and Shepherdson, J. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

Mancarella, P. and Pedreschi, D. 1988. An algebra of logic programs. In R. A. Kowalski and K. A. Bowen Eds., *Proc. 5th Int. Conference on Logic Programming* (1988), pp. 1006–1023. The MIT Press.

McCabe, F. 1992. *Logic and Objects*. Prentice Hall International, London.

Miller, D. 1989. A logical analysis of modules in logic programming. *Journal of Logic Programming 6*, 79–108.

Monteiro, L. and Porto, A. 1989. Contextual logic programming. In G. Levi and M. Martelli Eds., *Proc. 6th Int. Conference on Logic Programming* (1989), pp. 284–302. The MIT Press.

O'Keefe, R. 1985. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery Eds., *Proceedings of IEEE Symposium on Logic Programming* (1985), pp. 152–160. IEEE Computer Society Press.

Warren, D. 1983. An abstract Prolog instruction set. Technical Report TR 309, SRI International.