

060

LOGIC PROGRAMMING SYSTEMS USING GENERAL CLAUSES AND DEFAULTS

Ramiro Guerreiro, Andrea Silva, Marco A. Casanova

Rio Scientific Center - IBM Brazil

P.O. Box 4624

20.001, Rio de Janeiro, RJ - Brazil

RAMIRO at RIOVMSC

This paper describes two logic programming systems, with the expressive power of full first-order logic and a nonmonotonic component, which provide a direct generalization of pure Prolog and can be implemented using the same technology as Prolog processors. The inference engine of the systems is based on an extended version of the weak model elimination method enhanced with defaults.

LOGIC PROGRAMMING SYSTEMS USING GENERAL CLAUSES AND DEFAULTS

Ramiro Guerreiro, Andrea Silva, Marco A. Casanova

Rio Scientific Center - IBM Brazil

P.O. Box 4624

20.001, Rio de Janeiro, RJ - Brazil

RAMIRO at RIOVMSC

This paper describes two logic programming systems, with the expressive power of full first-order logic and a nonmonotonic component, which provide a direct generalization of pure Prolog and can be implemented using the same technology as Prolog processors. The inference engine of the systems is based on an extended version of the weak model elimination method enhanced with defaults.

1 MOTIVATION

1.1 Why do I need more than definite clauses and negation by failure?

We briefly discuss in this section three classes of logic programming systems, defined according to the type of logic programs and queries they accept. The discussion will help motivate our interest in logic programming systems based on weak model elimination (Loveland [1978]). Readers not familiar with Prolog are invited to skip this section.

A *clause* is a sequence of literals. A *program* is a finite set of clauses and a *query* is a disjunction of conjunctions of literals, that is, a quantifier-free formula in disjunctive normal form. A query is *definite* iff it is a single conjunction of literals, otherwise it is *indefinite*.

Recall that a *definite program clause* is a clause with exactly one positive literal and a *definite goal clause* is a clause with no positive literals. Usually a definite clause $L_0 \neg L_1 \dots \neg L_n$ is denoted by an expression of the form $L_0 \leftarrow L_1, \dots, L_n$, where L_0 is the *head* of the clause and L_1, \dots, L_n is the *body* of the clause, and a goal clause $\neg L_1 \dots \neg L_n$ is denoted by an expression of the form $\leftarrow L_1, \dots, L_n$.

We also need two other concepts. We define a *normal program clause* as an expression of the form $h_0 \leftarrow b_1, \dots, b_n$, where h_0 , the *head* of the clause, is a positive literal and b_1, \dots, b_n , the *body* of the clause, is a list of positive or negative literals. Finally, we define a *normal goal clause* as an expression of the form $\leftarrow b_1, \dots, b_n$ where b_1, \dots, b_n is again a list of positive or negative literals.

Now, we say that a logic programming system is *full first-order* if it accepts programs and queries as defined above. Section 2 of this paper will then discuss a specific full first-order system based on weak model elimination. We say that a system is a *pure Prolog* system if it accepts programs consisting of finite sets of definite program clauses and queries expressed by definite goal clauses. Finally, we say that a system is an *extended Prolog* system if it accepts programs consisting of finite sets of normal program clauses and queries expressed by normal goal clauses and, moreover, it treats negated literals by a special inference rule, called *negation by finite failure* (NFF) (Clark [1978]). Roughly, the NFF rule says to assume that p is false if one fails finitely to answer YES to the query $\leftarrow p$ in the presence of the program clauses.

We argue that in some cases full first-order systems are an interesting alternative to pure or extended Prolog systems. We first emphasize two points about full-first order systems:

- the knowledge representation language of such systems has the same power as full first-order languages and, in particular, it maintains the classical meaning of negation;
- they can be implemented efficiently using the technology developed for Prolog processors, especially those based on weak model elimination, as argued elsewhere (Casanova and Walter [1986a,b]).

The rest of this section stresses the importance of the first point.

Although pure Prolog systems have Turing machine power, the restriction to definite clauses makes it awkward to represent certain applications. The answer to this problem came with extended Prolog systems that treat negated literals by the NFF rule. We shall now show that this approach may in some cases be a worse alternative than returning to full first-order systems.

We may point out at least three important and distinct characteristics of the NFF rule:

- NFF is easy to implement in Prolog systems;
- NFF is a nonmonotonic rule justified on the grounds of the so-called "*Closed World Assumption*" (CWA) (Reiter [1978]);
- it is very difficult to give a theoretically precise characterization of NFF (that is, one for which NFF is sound and complete, see Lloyd [1986]).

The first characteristic is undoubtedly the greatest argument in favor of the adoption of the NFF rule.

The second characteristic can be taken either in favor or against the use of NFF, depending on whether the CWA holds or not for the application in question. If the CWA does not hold, an extended Prolog system will hardly be adequate and one should seriously consider a full first-order logic programming system.

The third characteristic can be best examined with the help of a very simple example. Consider the question of expressing a disjunction $p \vee q$ as a general program clause. If we naively take the symbol \leftarrow to mean (reverse) implication and \neg to mean true negation, then $p \vee q$ is indeed equivalent to $p \leftarrow \neg q$ and to $q \leftarrow \neg p$. But this equivalence is not true because negated literals are treated by the NFF rule. Indeed, let $P_1 = \{p \leftarrow \neg q\}$ and $P_2 = \{q \leftarrow \neg p\}$ be two programs. Let Q be the query $\leftarrow p$. Then, the answer of Q to P_1 will be TRUE and to P_2 will be FALSE in an extended Prolog system.

This apparently incorrect behavior will certainly shock naive Prolog users, but it can be explained by appealing, for example, to Clark's theory of program completion (see Clark [1984]). Indeed, denote by $\text{comp}(S)$ the completion of a program S . Then, $\text{comp}(P_1) = \{p \Leftrightarrow \neg q, \neg q\}$ and $\text{comp}(P_2) = \{q \Leftrightarrow \neg p, \neg p\}$. Hence, p is indeed a logical consequence of $\text{comp}(P_1)$ but not of $\text{comp}(P_2)$, which correctly *explains* the previous answers, but by all means *justifies* the bizarre behaviour of extended Prolog systems.

From this simple discussion, it becomes clear that, although NFF indeed extends in some sense the expressiveness of pure Prolog systems, it greatly complicates the theory to the point of raising serious questions even about soundness.

One last word should be said about disjunctions. In general, from a disjunction and a definite clause one may infer a clause as defined in beginning of this section. Hence, it requires a full first-order system to process logic programs consisting of finite sets of arbitrary disjunctions and definite clauses.

To summarize this last part of the discussion, quite expectedly, applications that require expressing disjunctive information, or some other form of information not easily captured by definite clauses, may be better modelled using full first-order logic programming systems.

1.2 May defaults help me?

AI researchers have set great effort to capture the ability of people to act "rationally" in the absence of complete, definitive knowledge about situations. Since knowledge of an agent about the world is necessarily incomplete, there will always be times when people will be forced to draw conclusions based on such incomplete knowledge. In such cases, assumptions and also hypothesis are made implicitly or explicitly. Of course these assumptions and hypothesis may have to be withdrawn at some later time should new evidence prove them invalid. If this happens, all conclusions based on those assumptions and hypothesis must also be withdrawn. This causes any system which attempts to reason consistently using assumptions and hypothesis to exhibit nonmonotonic behavior.

Default logic (Reiter [1980]) is an interesting approach to formalize nonmonotonic reasoning. Such approach allows deriving conclusions such as "the bird Tweety flies" based on a *default rule* or, simply, a *default*, like:

$$\frac{\text{bird}(x) : \text{fly}(x)}{\text{fly}(x)}$$

whose meaning is "if x is a bird and it is consistent to assume that x can fly, then infer that x can fly". Thus, in the presence of $\text{bird}(\text{Tweety})$, we may conclude $\text{fly}(\text{Tweety})$. Hence, from the default theory $\Delta = (\{D_1\}, \{\text{bird}(\text{Tweety})\})$ we may deduce $\text{fly}(\text{Tweety})$.

So defaults are a very elegant nonmonotonic inference scheme. They allow a precise way of choosing which predicate or sentence must be nonmonotonically inferred and what are the pre-conditions for that inference. In many cases defaults are an alternative to NFF. In some of these cases a more natural representation of the problem may be attained using defaults. Furthermore, from the theoretical point of view, defaults have a clear semantics.

2 A Short Description of two Logic Programming Systems

This section briefly describes two logic programming systems that extend standard Prolog. The first one, called STORK, is a full first-order system supporting classical negation as well as negation by finite failure. The second system, called PENGUIN, extends STORK with a special class of open defaults to capture nonmonotonic reasoning.

STORK also offers a collection of extra-logical commands, carefully designed to extend the meaning they have in Prolog, or designed to deal with programming aspects arising only in general clause logic programming. Among the commands borrowed from Prolog, we find the metapredicate " $\neg\%$ ", for negation by finite failure, the extra-logical predicate "cut" and the metapredicate "call", which in STORK also correctly handles indefinite answers. As an example

of a new command, we have the metapredicate "@", which is similar to "call" but computes only definite answers. To summarize, STORK extends both the declarative and the operational semantics of Prolog, including the extra-logical features, which means that a Prolog program retains its original meaning in STORK.

The inference engine of STORK is based on a version of the weak model elimination method. This refutation method offers an interesting alternative for the construction of logic programming systems since it accepts generic clauses, is input linear, does not use factoring but, in spite of these characteristics, maintains completeness. The search space can also be reduced by filters that restrict the application of the inference rules.

Defaults bring up new programming aspects, which demand special extra-logical predicates, incorporated in PENGUIN. For example, the metapredicate "def(X)" controls the use of defaults for answering queries. If X is instantiated to the atom "on" defaults are used, if it is instantiated to the atom "off" the use of defaults is forbidden. If X is free, then the current flag is returned. The refutation procedure for PENGUIN is an extension of weak model elimination along the lines of Reiter's default logic.

STORK and PENGUIN are both coded in VM/Prolog Release 1.0 (Program Number 5785-ABH) and are completely operational. The implementation started with the familiar idea of building Prolog interpreters in Prolog, adapted to the specific characteristics of model elimination. The Prolog code suffered successive optimizations until it achieved an adequate behavior in terms of space and memory. By using this strategy, the implementations could reuse all input-output and workspace management extra-logic commands of Prolog, among others. We just note here that the prototype for PENGUIN implements the consistency test required by the use of defaults through a strategy quite similar to that adopted to implement negation by finite failure.

To summarize, both STORK and PENGUIN provide more expressive power than Prolog. In particular, they enable the programmer to use classical as well as finite failure negation, choosing which one suits best the application. Moreover, PENGUIN also allows the representation of default information, that is, it permits carrying on nonmonotonic reasoning using defaults and, possibly, finite failure negation.

Finally, a detailed account of the theoretical foundations of the systems reported here can be found in Casanova et alii [1989a,b] and Guerreiro et alii [1989a,b]. Other attempts to extend Prolog to the expressive level of full first-order logic can be found in Lloyd and Topor [1985], Aida et alii [1983] and Poole and Goebel [1986]. A proposal for a non-clausal system with many inference rules is described in Bowen [1982]. Our approach differs from these in that it is a very natural extension of Prolog, without undue resort to extra-logical paraphrases. This was possible by a judicious choice of the basic refutation method and the adoption of defaults to capture nonmonotonic reasoning.

REFERENCES

- Aida, H., H. Tanaka and T. Moto-Oka [1983]. "A Prolog Extension for Handling Negative Knowledge", *New Generation Computing*, 87-91.
- Bowen, K.A. [1982]. "Programming with Full First-Order Logic", in *Machine Intelligence 10*, 421-440.

- Casanova, M.A., R.A.T. Guerreiro and A. Silva [1989a]. "Foundations of Logic Programming based on Model Elimination", North American Conf. on Logic Programming, Cleveland, Ohio (Oct. 1989) and also Technical Report CCR073, Rio Scientific Center, IBM Brazil (Apr. 1989).
- Casanova, M.A., R.A.T. Guerreiro and A. Silva [1989b]. "Logic Programming with General Clauses and Defaults based on Model Elimination", 11th Int'l. Joint Conf. on Artificial Intelligence, Detroit, Michigan (Aug. 1989).
- Clark, K.L. [1978]. "Negation as Failure", in *Logic and Databases*, H. Gallaire e J. Minker (eds.), Plenum Press.
- Guerreiro, R.A.T., Silva A. and Casanova, M.A. [1989a]. "Computing Answers in Default Logic", IEEE Int'l. Workshop on Tools for Artificial Intelligence, Fairfax, Virginia (Oct. 1989).
- Guerreiro, R.A.T., Silva A. and Casanova, M.A. [1989b]. "Foundations of Logic Programming with Defaults", (technical report in preparation).
- Lloyd, J.W. [1986]. *Foundations of Logic Programming*, Springer-Verlag.
- Lloyd, J.W. and R.W. Topor [1985]. "Making Prolog more Expressive", *J. Logic Programming* 2:2, 93-109.
- Loveland, D.W. [1978]. *Automated Theorem Proving: a Logical Basis*, North-Holland Publishing Company, Amsterdam.
- Poole, D.L. and R. Goebel [1986]. "Gracefully Adding Negation and Disjunction to Prolog", Proc. Third Int'l. Conf. on Logic Programming, London, UK (July 1986).
- Reiter, R. [1980]. "A logic for default reasoning", *Artificial Intelligence* 13, 81-132.

APPENDIX - Example and Performance Measurements

All performance data presented in this appendix are approximated values.

Consider the following program (which cannot be easily coded in Prolog):

```
man(marcus).
pompeian(marcus).
~pompeian(X) | roman(X).
ruler(caesar).
~roman(Y) | loyalto(Y,caesar) | hate(Y, caesar).
loyalto(Z,f(Z)).
~man(W) | ~ruler(W1) | ~tryassassinate(W,W1) | ~loyalto(W,W1).
tryassassinate(marcus,caesar).
```

STORK took 20 ms and, on the average, 504 bytes of Prolog global stack, 600 bytes of Prolog local stack and 40 bytes of Prolog trail stack to answer the query: `hate(marcus,X)`. Note that the above statistics includes the meta-interpreter overhead and, hence, is not just a direct consequence of the use of general clause or the refutation method STORK uses. As a measure of the meta-interpreter overhead, we note that the standard list inversion algorithm took 2 ms to invert a list of 52 elements, when ran on VM/Prolog in our system, and 3400 ms, when coded and ran on STORK. This comparison is meaningful because, for definite clauses, the refutation method STORK uses degenerates into that of Prolog.