A Metalogic Programming Language

Stefania Costantini
Gaetano Aurelio Lanzarone

Universita' degli Studi di Milano,
Dipartimento di Scienze dell' Informazione
Via Moretto da Brescia 9, I-20133, Milano, Italy
Tel. +39-2-2772.222, Telex 335199 MI DSI I,
e-mail: logic@imisiam.bitnet

Abstract

A language called Reflective Prolog is presented, aimed at moving a step forward in the declarative representation of metaknowledge within the logic programming approach. Reflective Prolog is a metalogic programming and knowledge representation language which allows the definition of object-level and metalevel sentences, both expressed in Horn clauses extended with self-reference and metaevaluation capabilities. Namely, language entities can be described in the language itself, in order to represent metaknowledge, and a meta-evaluation level can be defined in a program, in order to express auxiliary deduction rules. Metasentences are processed by the language interpreter via an extended resolution procedure, with an efficiency comparable in principle to that of a Prolog interpreter. After stating the main objectives of the work and the relationship with the literature, the syntax and the procedural semantics of the language are described; how the model-theoretic semantics has been defined is also mentioned. An example is then discussed in order to illustrate the capabilities of Reflective Prolog. Since the interpreter of the language has been implemented, the architecture of the implementation, based on reflection mechanisms, is outlined. Directions of further reasearch are sketched in the conclusions.

Introduction

The importance of metaprogramming and metaknowledge representation is well known in the Logic Programming and in the Artificial Intelligence areas (see for instance [1]). The Logic Programming

community has recognized and practically exploited (especially with the technique of metainterpreters) Prolog's features in this direction since its very beginning, and is in search of a systematization for a logic metaprogramming language since the seminal work of [3].

The present situation seems to be the following [17]. On the one hand, the implemented languages, like Metalog, lack a semantic definition, though a first attempt has been made very recently [23]. On the other hand, work is in progress to rationalize the metalogic features of Prolog and to give a precise meaning to metaprograms [11], in view of the definition of a suitable language.

In this paper we present a metalogic programming language which we have both syntactically and semantically defined [7], and fully implemented in a working prototype [2]. The objectives of our work, and the assumptions on which we have based our approach, are the following.

1. The main objective is to improve the declarative expressive power of a logic programming language. We also agree with Gallaire's claim [9] that in order to gain ground in the real world, the logic programming approach has to provide more effective languages and tools. We believe that this is necessary especially for sophisticated applications like those in the Artificial Intelligence area, which is still dominated by Lisp and other formalisms. Some applications of the problem–solving capabilities of the language we are proposing have been shown in [5].
2. It is advisable to maintain the language characteristics within computationally tractable bounds, amenable in principle to efficient implementations. We stayed within a first–order, Horn–clause, resolution–based language.
3. It is important to achieve the added power of the language not at the expense of its logic basis. We gave up altogether the 'impure' metalogic predicates of Prolog and defined differently the metastructural features of the language [4]. Specifically, the language has been equipped with self–reference capabilities based on a full naming mechanism, similar to those of [19], and with a multilevel structure. The latter consists of a base (object and meta) execution level and a metaevaluation level; switching between levels is taken care of by an extended resolution algorithm, and has Feferman's logic reflection principles [8] as the semantic counterpart.

[4] The realization of the language should not simply be an implementation exercise, instrumental in obtaining the interpreter, but has to be based on a suitable computational architecture. We identified procedural reflection [21], [10] as the simple and powerful technique best fitting the deductive apparatus of the language.

1. The Language

Syntax and semantics of Reflective Prolog are fully defined in [7]; they are described here only by mentioning the main differences with respect to the usual definitions of definite programs [15].

The language defined by a Reflective Prolog program P consists of all the well–formed formulae in Horn–clausal form obtained from the alphabet of P. The main difference in the alphabet with respect to traditional Horn–clause languages is the presence of *metavariables* and *name constants*. Each metavariable is either a *function* metavariable (written with first character '$') or a *predicate* metavariable (written with first character '#'). Each name constant is either a *quoted* name constant (written enclosed in quotation marks) or a *bracketed* name constant (written enclosed in angle brackets). The alphabet contains the distinguished function symbols predication, predicate, function, functor, arity, args, and the distinguished unary predicates solve, theory_solve and theory_fact.

The definition of terms is extended to deal with self–reference: variables can be either object variables or metavariables, and there is a new class of terms, *name terms*, that are defined inductively as follows.

- *Name constants* (either quoted or bracketed), described above.
- *Function name terms*, which are of the form function(functor(ϕ), arity(I), args(α_1, ..., α_n)), where ϕ is a quoted name constant, I is a non–negative integer number and (α_1, ..., α_n) are name terms. The following abbreviations are available:
  - "ϕ(α_1, ..., α_n)", where quotation is shifted from inside to outside the term and arity is omitted; e.g., "f(a)" is shortened for function(functor("f"), arity(1), args("a")).
  - "ϕ"(α_1, ..., α_n), "f"("a") for the above example, where arguments are extracted and arity omitted.

- *Relation name terms*, which are of the form predication(predicate(ρ), arity(I), args(τ_1, ..., τ_n)), where ρ is a bracketed name constant, I is a non–negative integer number and (τ_1, ..., τ_n) are name terms. Abbreviations similar to those for function name terms are allowed, that is "ρ(τ_1, ..., τ_n)" (e.g., predication(predicate(<p>), arity(1), args("a")) can be shortened as "p(a)") as well as <ρ>(τ_1, ..., τ_n) (<p>("a")).

Name terms realize the ground representation of Reflective Prolog terms and atoms in the language itself: the definitions above introduce quoted name constants to denote constants, variables and function symbols (e.g. "c", "V", "fun"); brackteted name constants to denote predicate symbols (e.g. <pred>); function and relation name terms to denote terms and atoms respectively (e.g. "fun"("c","V") and <pred>("c","V"), which are sintactically equivalent to "fun(c,V)", "pred(c,V)").

Name terms are internally represented in the language interpreter in the form <ρ>(τ_1, ..., τ_n), because subterms are directly accessible, allowing easy inspection and composition/decomposition of metalevel terms (with no need of either run–time parsing or complex term manipulation, that a less expressive representation would require). Conversion among the different available forms is taken care by the language parser at program–consultation time.

Terms and atoms can be seen as divided into *metalevel terms (atoms)*, containing name terms and/or metavariables as subterms, and *object–level terms (atoms)*.

The notion of Reflective Prolog *definite clause* includes some syntactic restrictions. First, levels are distinct: if the conclusion of a clause is an object–level atom, the conditions must be object–level atoms as well and, conversely, if the conclusion is a metalevel atom, the conditions must be metalevel atoms. Moreover, being the language based on clear distinction between use and mention, a syntactic prevention against auto–mention is provided: atoms such as p(<p>), containing a reference to their own head predicate among the arguments, are not allowed.

A Reflective Prolog definite program is divided into a *metaevaluation level*, consisting of the clauses defining the distinguished predicate solve and its auxiliary predicates, and a *base level*, consisting of the remaining metalevel clauses and all the object–level clauses. Precisely, a predicate p belongs to the metaevaluation level if:

(i) its definition makes use of at least one of the distinguished predicates solve, theory_solve, theory_fact;

(ii) it is directly or indirectly used in the definition of solve; that is, it either appears in the body of a solve clause, or it is directly or indirectly used in the definition of some predicate q that appears in the body of a solve clause.

The distinguished predicates can only be used in clauses at the metaevaluation level, and cannot be nested, not even one within the other. Metaevaluation predicates cannot be used at the base level. Conversely, base-level predicates can be used at the metaevaluation level, provided that the above-mentioned syntactic restrictions are respected. All syntactic restrictions are verified at program-consultation time.

The unification algorithm is extended to deal with metavariables and metalevel terms. Namely, rules for substitution are:

[1] any object-level variable V can be substituted by an object-level term t distinct from V;

[2] any predicate metavariable #V can be substituted by a bracketed name constant <t>;

[3] any function metavariable $V can be substituted by a metalevel term t distinct from $V.

The extended unification algorithm has been proved to terminate finitely, giving either an mgu for the expressions to be unified, or reporting that they are not unifiable [7]. Relying on a flexible representation of metalevel terms, extended unification can be easily implemented as a straight extension of object-level unification, with still linear complexity and only a slight overhead for the above-specified "type-checking" in variable substitution.

Derivation by resolution is extended to state the role of base (object- or meta-) level clauses as well as metaevaluation clauses in a proof. Forms of reflection are introduced to shift from the base level to the metaevaluation level and vice versa. The definition is the following, where the notation ↑A is adopted, to indicate the relation name term denoting the atom A (for instance, ↑p(a) stands for <p>("a")).

Definition. (Reflective SLD–Resolution, or RSLD–Resolution)

Let G be a definite goal $\leftarrow A_1,\ldots,A_m,\ldots,A_k$ and C a clause. If A_m is the selected atom in G, then G' is derived from G and C using mgu θ iff one of the following conditions holds:

(i) C is $A\leftarrow B_1,\ldots,B_q$
θ is an mgu of A_m and A
G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$

(ii) C is $solve(\alpha)\leftarrow S_1,\ldots,S_w$,
$A_m \neq solve(\delta) \wedge A_m \neq theory\_solve(\delta) \wedge A_m \neq theory\_fact(\delta)$
θ is an mgu of $\uparrow A_m$ and α
G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},S_1,\ldots,S_w,A_{m+1},\ldots,A_k)\theta$

(iii) A_m is solve(M)
C is $A\leftarrow B_1,\ldots,B_q$
θ is an mgu of M and $\uparrow A$
G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$

(iv) A_m is theory_solve(M)
C is $A\leftarrow B_1,\ldots,B_q$
θ is an mgu of M and $\uparrow A$
G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$

(v) A_m is theory_fact(M)
C is $A\leftarrow$
θ is an mgu of M and $\uparrow A$
G' is the goal $\leftarrow(A_1,\ldots,A_{m-1},A_{m+1},\ldots,A_k)\theta$

To explain, the definition considers the following different possibilities about the selected goal A_m.

[1] $A_m \neq$ solve(δ), theory_solve(δ), theory_fact(δ). The goal A_m can be proved in two ways. First, (similar to classical SLD–Resolution) using the clauses defining the corresponding predicate (case (i)): for instance, if A_m = p(a,b), the clauses defining the predicate p, whose head unifies with p(a,b). Second, the clauses defining the solve predicate, whose head argument unifies with $\uparrow A_m$, can be used (case (ii), *upward reflection*): in the above example, clauses with head solve(<p>("a","b")), or solve(<p>($X,$Y)) or solve(#P("a",$K)), and so on.

[2] A_m = solve(δ). Again, there are two choices. First, using the clauses defining the predicate solve itself, similarly to any other goal (case(i)). Second, using the clauses defining the predicate corresponding to the atom denoted by the argument δ of solve (case (iii), *downward reflection*): for instance, if A_m = solve(<q>("X","b")), the clauses defining the predicate q, whose

head unifies with q(X, b). That is, each goal generated at the metaevaluation level can be tried at the metaevaluation level, as well as at the base level.

[3] A_m = theory_solve(δ). The clauses defining the predicate corresponding to the atom denoted by δ can be used (case(iv), *downward reflection*): if for instance A_m = theory_solve(<h>("k")), the clauses defining the predicate h, whose head unifies with h(k). That corresponds to proving at the base level the (dereferenced) argument of theory_solve.

[4] A_m = theory_fact(δ). A fact is searched at the base level that matches the (dereferenced) argument of theory_fact. For instance, theory_fact(#P("a")) matches the representation <h>("a") of a fact h(a), instantiating #P to <h>.

The resolution procedure implemented in the Reflective Prolog interpreter follows a generalized depth–first strategy, where cases (i) and (iii) have higher priority than (ii): each goal is first attempted at the base level and then at the metaevaluation level. It is worth noting that, given a selected atom A_m in a definite goal G, the new selected atom A_m' in the new goal G' obtained as described above is treated in the same way: that is, an *RSLD–Refutation* is in general obtained by an interleaving between base level and metaevaluation level.

The following is a simple example of a Reflective Prolog program.

```
/* metaevaluation level */
    solve(#P($X,$Y)):–symmetric(#P),theory_solve(#P($Y,$X)).
/* base level */
    /* metalevel */
    symmetric(<friend>).
    /* object–level */
    friend(giorgio,mary).
    friend(lucia,albert).
```

The solve rule defines symmetry: (the objects denoted by) $X and $Y are in the relation (denoted by) #P in the theory, provided that the theory contains the assertions of #P being symmetric and it can be proved that $Y and $X are in #P.

Queries can concern both object–level and metalevel predicates. For instance, the query ?–symmetric(#P) instantiates #P to <friend>; the query ?–friend(albert,lucia) fails at the base level, causing the application of the solve rule, which first proves symmetric(<friend>) and

then attempts theory_solve(<friend>("lucia","albert")), determining a shift-down to friend(lucia,albert) that succeeds.

This formulation avoids the problem of circular application of symmetry when expressed at the object-level, since the call to theory_solve results in directly proving its argument at the base level; in addition, exclusion of the metaevaluation rule when it is no longer needed significantly improves efficiency.

At this point, a comparison with other approaches is in order.

The distinguished predicate solve is not an axiomatization of provability, but expresses only what is beyond the normal behaviour. The distinguished predicate theory_solve stands for provability in the theory, but it is engendered by the interpreter rather than explicitly defined: it is a way for a theory to refer implicitly to its own inference mechanism.

Reflective Prolog is not an amalgamated language in the sense of [3]: the provability relation is not explicitly expressed in the language, and it is not the case that no theorem is provable that cannot be proved either in the language or in the metalanguage. The different levels are distinct, but procedurally and semantically integrated. The flexibility and power of their interaction through the extended resolution makes new theorems provable, which cannot be proved in the base-level or in the metaevaluation level only.

The metaevaluation level of Reflective Prolog is basically different from a Prolog metainterpreter, for the following reasons. First, base-level resolution need not be reproduced at the metaevaluation level, with the consequent gain in efficiency and conciseness. Second, different metaevaluation rules can be easily integrated, relying on the fact that each goal is metaevaluated when needed. On the contrary, with a Prolog metainterpreter it is necessary to explicitly state which goals are to be metaevaluated and which ones can be directly evaluated: possible uses of metaknowledge in a proof must therefore be identified in advance. Third, since base clauses and metaevaluation clauses are treated in the same way by the interpreter, there is no run-time overhead related to upward and downward reflection (other than referencing-dereferencing a goal). Fourth, the interpreter performs run-time checks to prevent auto-reference, forcing immediate failure of calls, such as p(<p>), that could be attempted via backtracking on the solve rules, leading to infinite loops; solving the same problem in Prolog metainterpreters is not at all straightforward.

To summarize, Reflective Prolog provides an automathic ("when needed") form of reflection (considered an open problem in [16]). This overcomes the "non insignificant control problem" [3] of when to use reflection, that contrasts with the principle of expressing knowledge in declarative form and leaving the procedural aspects to the deductive apparatus of the language [13]. Augmenting resolution with implicit reflection leads to reintroduce a declarative attitude in the definition of metaknowledge, without sacrificing efficiency.

The declarative semantics of the language has been defined in a model–theoretic fashion: a least model semantics has been provided, based on the definition of a concept of *Least Reflective Herbrand Model* of a theory. Starting from a notion of Extended Herbrand Interpretation of a Reflective Prolog definite program P (introduced to deal with name terms and distinguished predicate symbols), a Reflective Herbrand Model for P is defined to be an Extended Herbrand Interpretation which is a model for the theory P' obtained by applying to P the first of the *Reflection Principles* introduced in [8]. The Least Reflective Herbrand Model has then been characterized (similarly to the classical Horn–clause language semantics [15]) as the fixpoint of a suitable mapping, in order to provide a link between the declarative and procedural semantics of a Reflective Prolog definite program.

2. An example

The declarative representation of the transitivity property of a binary relation in Prolog:

```
r(X,Y):-r(X,Z),r(Z,Y).
```

cannot be procedurally used without problems, since with the depth–first strategy the interpreter may undergo an infinite branch before examining success branches. The problem is the application of the recursive clause twice consecutively: it has to be interleaved with the application of different clauses (facts); using a flag for this purpose [18] is awkward. Although in some cases a transitive relation can be defined in terms of transitive closure of a subrelation, this is not feasible for every relation. Even for those relations that can be defined as closures, it is not always possible to represent all the intended connections. Given for example the clauses:

```
parent(a,b).
parent(c,d).
```

```
ancestor(b,c).
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
```

the solutions (a,d) and (b,d) to the query ?- ancestor(X,Y) cannot be obtained unless the following clause is added to the database (determining repetitions in answers):

```
ancestor(X,Y):-parent(Z,Y),ancestor(X,Z).
```

But also in this case, adding the fact ancestor(c,e) the same query cannot obtain the answers (a,e) and (b,e).

The problem occurs not only with Prolog, but also with other representation and inference systems [20]. One solution to determine the couples belonging to a transitive (as well as symmetric and/or reflexive) relation, is that of representing the relation couples by means of composable sets (as in [20]), and defining a suitable metainterpreter which uses them to infer those couples not explicitly asserted [6].

A simpler solution, which is possible in Reflective Prolog, is the following.

```
/* metaevaluation level */
solve(<ancestor>($AL)):-fact(<ancestor>($AL)).          [1]
solve(<ancestor>($X,$Y)):-fact(<ancestor>($X,$Z)),      [2]
                           solve(<ancestor>($Z,$Y)).

fact(<ancestor>($AL)):-theory_fact(<ancestor>($AL)).
fact(<ancestor>($AL)):-theory_fact(<parent>($AL)).

/* base level */
parent(a,b).
parent(c,d).
ancestor(b,c).
ancestor(c,e).
```

Rules [1] and [2] define transitivity of the relation ancestor in a way that prevents loops by forcing each transitive step to be performed on facts; the predicate fact gives the possibility of considering both ancestor and parent facts.

For instance, the query ?- ancestor(a,e) is proved by means of subsequent applications of rule [2] and a final application of rule [1], precisely in the following way.

- On failure of the goal at the object level, the consequent shift–up leads to selection of rule [1] that fails because neither the fact ancestor(a,e) nor the fact parent(a,e) exist in the database. The extended unification algorithm provides a special feature for name terms, i.e. it is able to unify <ancestor>("a","e") and <ancestor>($AL) by binding $AL to ["a","e"].
- Selection of rule [2] results in attempting
  fact(<ancestor>("a",$Z)), solve(<ancestor>($Z,"e")).
  The first subgoal succeeds via the second alternative of fact, that is
  fact(<ancestor>("a",$Z)):–theory_fact(<parent>("a",$Z))
  that instantiates $Z to "b".
- Then solve(<ancestor>("b", "e")) is attempted, leading in a similar way to solve(<ancestor>("c","e")) that succeeds via rule [1] and the first alternative of fact, concluding the proof.

To further develop the example, it can be noted that a family has actually a tree structure, where ancestors are connected to descendants by paths (a *metaphor* can be stated between the two concepts, to follow the terminology of [12]). In fact, the definition of path–finding in a generic directed acyclic graph, shown below, is appearently analogous to ancestor determination.

```
solve(<path>($AL)):–fact(<path>($X,$Y)).                    [1]
solve(<path>($X,$Y)):–fact(<path>($X,$Z)),
                      solve(<path>($Z, $Y)).                [2]
fact(<path>($AL)):–theory_fact(<path>($AL)).
fact(<path>($AL)):–theory_fact(<arc>($AL)).
```

Hence, ancestor finding could also be obtained by adapting path finding, adding the following part:

```
solve(#P($AL)):–map(#P,#Q),solve(#Q($AL)).                  [3]
solve(#P($X,$Y)):–symmetric(#P),
                  theory_solve(#P($Y,$X)).                  [4]
map(<parent>,<arc>).
map(<ancestor>,<path>).
symmetric(<map>).
parent(a,b).
parent(c,d).
ancestor(b,c).
ancestor(c,e).
```

Suitable facts map family concepts into tree concepts (and vice versa, in that map is declared as symmetric), and metaevaluation rule [3] engenders the mapping. Rule [4] concerning symmetry is the same as in the previous section.

Notice that composition of different solve clauses is obtained simply by justaposition, while composition of metainterpreters needs elaborate techniques [22].

Rather than mapping one solution onto the other, a common general structure can be identified, expressed by the rules below.

```
solve(#P($AL)):-fact(#P($AL)).                                    [1]
solve(#P($X,$Y)):-transitive(#P),fact(#P($X,$Z)),
                        solve(#P($Z,$Y)).                         [2]
fact(#P($AL)):-theory_fact(#P($AL)).
fact(#P($AL)):-subsumes(#P,#Q), theory_fact(#Q($AL)).
```

Rules [1] and [2] define in general the transitivity of a relation, relying on the explicit declaration of a predicate being transitive. fact takes subsumption (intended as set inclusion of relation extensions) into account in facts lookup, stating that if a predicate #P subsumes another predicate #Q, then #P($AL) (where $AL denotes the argument list) holds in the theory if #Q($AL) holds. It also relies on explicit declaration of subsumption.

Supposing that the above theory is available, for instance in a system library, ancestor determination as well as path-finding can be obtained by adding to that theory a suitable definition of the base level, containing the object facts plus metalevel declarations of transitivity (for ancestor and path respectively) and subsumption (of parent by ancestor and of arc by path). In the family case for instance, the base level to be added is the following.

```
subsumes(<ancestor>,<parent>).
transitive(<ancestor>).
parent(a,b).
parent(c,d).
ancestor(b,c).
ancestor(c,e).
```

In substance, the general theory of transitivity constitutes a metaphor for both families and directed acyclic graphs powerful enough not to require to be adapted, but only to be used. Put aside the problem of how to recognize such abstractions, in Reflective Prolog it is possible

to express and make use of them.

The Reflective Prolog interpreter also has the ability to expand and include in the data base some consequences of the program during program consultation, guided by directives expressed as metasentences. In the example, a directive transitive_closure(#P) can be defined (and then called over <ancestor>), that expands the definition of its argument, adding to the database all the facts obtained as transitive closure from those already present. In this case, any subgoal concerning ancestor is solved in just one step. Trading off time against space is left to the user; in both cases, expressiveness and conciseness in theory definition are not affected.

3. The implementation

Reflective Prolog has been implemented on the basis of a Horn clause language with procedural reflection, formerly developed (in Quintus Prolog on the IBM 6150 workstation) in our Department.

The procedural reflection architecture allows program execution to develop (on demand) along an arbitrary number of levels; at each level, the deductive apparatus of the basic formalism (in this case Horn clauses) is fully available. Shifting by procedural reflection to the next upper level, a program can look at and/or modify its own computational state. The state is explicitly modeled as the representation of the database and the proof tree (the latter including the binding environment) "frozen" at the moment of the shift–up.

Procedural reflection is a too low–level concept for direct use in applications, but is useful as an implementation tool for more advanced formalisms. It has been used for several features, like inspection and modification of variable bindings, necessary for the implementation of basic primitives of Reflective Prolog, namely extended unification and referentiation/dereferentiation. Furthermore, control predicates (like 'cut', 'fail') have been realized by procedural reflection, via access to the control part of the model (proof tree). These predicates do not have a high–level counterpart in Reflective Prolog, being related to aspects that are not explicit in the language.

Prolog's metalogic predicates for term inspection/manipulation are of course unnecessary in Reflective Prolog since explicit representation of terms and atoms is available, only the syntactic identity '==' is retained, realized via unification as in Prolog.

A prototype of the Reflective Prolog interpreter is actually working

[2], and includes a parser of the language (realized with the DCG approach), for translating programs and goals into a suitable internal form.

4. Conclusive remarks

In this paper we have presented Reflective Prolog, a metalogic programming language. The main objective was to show how better expressiveness, flexibility and computational power can be achieved in the context of a Horn–clause, resolution–based language with a firm semantic ground: syntactically, by introducing self–reference capabilities; computationally, by integrating forms of reflection in derivation.

The possibility is being investigated of making some aspects of the inference process explicit, so as to allow high–level forms of control (thus eliminating the need of predefined predicates like 'cut', 'fail'). The difficult is not in technical feasibility, but in the identification of: (i) what aspects to make explicit, and how to cope with them semantically; (ii) in which form they should be represented in a program, so as to preserve conceptual clarity, and to allow easy manipulation.

One theoretical open problem concerns the introduction of negation, in a semantically sound way. Recent investigations suggest [14] that the availability of a distinguished predicate, like solve, standing for provability, could help safely introducing 'negation as failure' in the language.

Presently, Reflective Prolog has no construct for the representation and manipulation at the metalevel of object–level clauses. Work is under way to define constructs able to include in the language the representation and use of several theories. However, in our opinion a proper use of theories is mainly related to hypothetical reasoning or reasoning involving multiple agents; in fact we think that the Reflective Prolog multilevel structure overcomes in other cases the need of using theories that arises in formalisms lacking this structure.

We are also carrying on studies and experiments on applying Reflective Prolog to sophisticated problem solving and intelligent reasoning. For example, some forms of non–monotonic reasoning seem amenable to elegant representation, but more work is needed to present results.

Acknowledgements

This work has been partly supported by IBM Italia S.p.A in the context of a joint study with the Computer Science Department of the University of Milan, and partly by the Italian Ministry of Public Education.

References

[1] L. Aiello, G. Levi, *The Uses of Metaknowledge in AI systems*, in: P. Maes and D. Nardi (eds.), Meta-level Architectures and Reflection, North-Holland 1988, 243-254.

[2] R. Barki, G. Casaschi, S. Costantini, P. Dell'Acqua and G.A. Lanzarone, *The Implementation of Reflective Prolog*, Internal Report, University of Milano, Computer Science Department, 1989.

[3] K. A. Bowen, R. A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming*, in: K.L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic Press, 1982, 153-172.

[4] S. Costantini, G. A. Lanzarone, *Towards Metalogic Programming*, in: Proceedings of *Computational Intelligence 88*, Milano, September 26-30, 1988, 41-52 (in course of publishing by North-Holland).

[5] S. Costantini, G. A. Lanzarone, *Problem Solving in Metalogic Programming*, in: Proceedings of IPCCC 89 - IEEE *Eighth Annual International Phoenix Conference on Computers and Communications*, Phoenix, Arizona, March 22-24, 1989.

[6] S. Costantini, G. A. Lanzarone, *On the Properties of Relations in Prolog and Beyond*, Internal Report n. 32/88, University of Milano, Computer Science Department, 1988 (appeared in italian in: Proceedings of AICA Conference, 1988).

[7] S. Costantini, *Formal Definition of Reflective Prolog*, Internal Report n. 49/89, University of Milano, Computer Science Department, 1989.

[8] S. Feferman, *Transfinite Recursive Progressions of Axiomatic Theories*, Journal of Symbolic Logic, n. 27, 1962, 259-316.

[9] H. Gallaire, *Boosting Logic Programming*, (invited talk) in: J.-L. Lassez (ed.), Proceedings of the Fourth International Conference on Logic Programming, MIT Press 1987, 962-988.

[10] R. Ghislanzoni, L. Spampinato and G. Tornielli, *Reflection as a Tool for Integration: an Exercise in Procedural Introspection*, Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milano, August 23-28, 1987, 44-47.

[11] P.M. Hill, J.W.Lloyd, *Analysis of Meta-programs*, in: [17], 27–42.

[12] B. Indurkhya, *Constrained Semantic Transference: A Formal Theory of Metaphors*, in: A. Prieditis (ed.), *Analogica*, Pitman 1988, 129–157.

[13] R. A. Kowalski, *Logic for Problem Solving*, North-Holland Elsevier, New York, 1979.

[14] V. Lifschitz, *Negation as Failure and Introspective Reasoning*, Invited Talk, Third International Symposium on Methodologies for Intelligent Systems, Torino, Italy, October 12–15, 1988.

[15] J.W. Lloyd, *Foundations of Logic Programming*, (2nd Ed.), Springer-Verlag, 1987.

[16] P. Maes, *Introspection in Knowledge Representation*, Proceedings of the 7th European Conference on Artificial Intelligence, Brighton, UK, 1986.

[17] Meta88, *Proceedings of the Workshop on Meta-programming in Logic Programming*, Bristol, June 22–24, 1988.

[18] D. Nute, *A Programming Solution to Certain Problems with Loops in Prolog*, ACM SIGPLAN Notices, vol. 20, n. 8, August 1985, 32–37.

[19] D. Perlis, *Languages with Self-Reference I: Foundations*, Artificial Intelligence, vol.25, 1985, 301–322.

[20] S.C. Shapiro, *Symmetric Relations, Intensional Individuals, and Variable Binding*, Proceedings of the IEEE, vol. 74, n.10, October 1986, 1354–1363.

[21] B. C. Smith, *Reflection and Semantics in LISP*, Xerox PARC ISL-5, Palo Alto, 1984.

[22] L. Sterling, A. Lakhotia, *Composing Prolog Meta-Interpreters*, Proceedings of the 5th International Logic Programming Conference, Seattle, 1988.

[23] V.S. Subrahmanian, *Foundations of Metalogic Programming*, in: [17], 53–66.