

# Reflection Mechanisms for Combining Prolog Databases

EVELINA LAMMA, PAOLA MELLO AND ANTONIO NATALI

*Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale  
Risorgimento 2, 40136 Bologna, Italy*

## SUMMARY

By using practical examples, this paper outlines the power of reflection mechanisms for logic programming systems in the domain of knowledge structuring. In particular, it presents an extension of Prolog, where separate databases can be handled as first-class objects. Different forms of database combination such as inheritance and dynamic context extension/contraction are specified and implemented in a dynamic and flexible way through reflection. The main aim is to broaden the application area of logic programming to encompass most of the paradigms needed by systems that use artificial intelligence techniques. Practical results presented in the paper show that logic programs that use reflection can be shorter, more readable and efficient than those using more conventional full meta-interpretation techniques. Full meta-interpretation, however, is more general than reflection.

KEY WORDS Logic programming Knowledge representation Meta-programming Reflection Prolog

## INTRODUCTION

Several researchers argue that logic programming, and Prolog<sup>1</sup> in particular, are not suitable to represent and manipulate knowledge. Indeed, essential features such as structuring and combining of large amounts of knowledge are difficult to handle in current logic-programming systems.

For this reason, extensions to Prolog have been proposed in the literature.<sup>2</sup> The aim of these proposals, including the one presented in this paper, is to provide a more powerful logic-programming language for complex practical applications, e.g. those that use methods of artificial intelligence, while fully preserving logical semantics.

It is widely recognized that a logic-based system for knowledge programming must allow the partition of programs into separate databases. Separate (Prolog) databases have been called modules,<sup>3</sup> worlds,<sup>4</sup> theories,<sup>5</sup> units,<sup>6</sup> etc., emphasizing different specific semantic issues. However, there is still no general agreement on how separate databases should be combined. Inheritance (simple or multiple), delegation and viewpoints are examples of database combination policies. They, have been introduced in several languages in order to enhance knowledge reusability and flexibility or to allow hypothetical and non-monotonic reasoning. The main drawback of these languages arises from the impossibility of extending their built-in set of knowledge-combination policies. Extensions can be made only by modifying the underlying virtual machine.

It is most widely believed that the logic-programming community needs experience in combining knowledge and exploring its capabilities before fixing specific policies

0038-0644/91/060603-22\$11.00

*Received 9 September 1988*

© 1991 by John Wiley & Sons, Ltd.

*Revised 12 July 1989 and 18 January 1991*

in a general purpose language. For this reason, meta-programming techniques are currently being fruitfully employed. Systems built by using meta-programming techniques are open-ended: it is possible to extend and change the underlying machine dynamically within the system itself.

The most widespread and straightforward technique for building systems based on meta-programming techniques in logic is to write a full (Prolog) meta-interpreter.<sup>7,8</sup> Meta-interpreters explicitly reproduce each part of the underlying machine that is to be extended or changed.

An alternative approach to providing meta-programming capabilities is to allow direct access to specific parts of the underlying machine state, i.e. reflection.

The main advantage of reflection techniques is that the writing of a full meta-interpreter can be avoided. The price to be paid for this is that meta-programs have a more limited scope, depending on the selected visibility of the underlying machine. Nevertheless, it is often possible to establish a better trade-off between efficiency and extensibility. In addition, meta-programs are more concise and readable.

In spite of these attractive features, reflection has had a relatively limited diffusion in the logic-programming community and is still poorly understood. The most relevant programs have been developed in LISP.<sup>9</sup>

The main aim of the paper is to face the problem of separate database composition by means of reflection.

We present an extended and reflective Prolog system that supports separate databases called units. The system provides a self-description of its computational state that can be manipulated directly without writing a full extended Prolog meta-interpreter. No predefine predicate need be introduced to combine units: their combination policies can be described in logic, using reflection mechanism<sup>7</sup> as a sort of high-level specification, without changing the basic code for the Prolog interpreter. The system (called CPU—Communicating Prolog Units<sup>6</sup>), has been helpful in elucidating design choices and facilitating comparative experiments on different schemes of combining units.

A second aim of the paper is to present and discuss different policies of knowledge structuring. In particular, context extension/contraction and inheritance policies are introduced and integrated by means of meta-programs.

A third part of the paper focuses on comparisons between reflection and full meta-interpretation techniques. The points to be considered in the comparison are

- (a) flexibility and generality
- (b) clarity, elegance, conciseness and readability
- (c) performance.

This paper stresses performance considerations. We make this emphasis because one of the main drawbacks of meta-programming techniques is the overhead that they introduce, and improvement in efficiency is necessary to make meta-programming a useful technique for implementation of real systems based on the logic-programming paradigm. Through a study of unit combination, this paper shows how the form of reflection used in CPU improves the trade-off between flexibility and efficiency.

#### AN EXTENDED REFLECTIVE PROLOG SYSTEM

In CPU, two basic features are added to standard Prolog: modularity and reflection mechanisms.

---

## Modularity

In standard Prolog programs a single database is present, but several extended Prolog interpreters already provide the notion of separate databases (see Prolog/KR,<sup>4</sup> MProlog,<sup>3</sup> ESP,<sup>10</sup> etc). In CPU, too, separate databases (called units) can be defined. Each unit is identified uniquely by a name (a Prolog constant) and is composed of a set of Horn clauses. Clauses whose heads have the same arity and functor, but belonging to different units, are considered as separate procedures. The interpreter tries to solve the current (sub-)goal by using only the clauses of the unit that happens to be current. The current (sub-)goal and unit are represented by two virtual registers of the interpreter: CG and CU, respectively.

Another auxiliary register called AUX has been added to the standard Prolog machine. Whereas CU and CG have specific meanings for the interpreter, no semantic meaning is associated *a priori* with the AUX register. It is up to the reflective system to handle the new AUX register properly so that new abstractions can be defined in the logic. The CU, CG and AUX registers represent the selected state of the CPU virtual machine which is directly accessible through reflection mechanisms. They are saved automatically in the current environment on the local stack and restored according to standard Prolog-machine behaviour.<sup>11</sup>

## Reflection

It is widely recognized that meta-programming<sup>2,7</sup> is a powerful and expressive technique for overcoming some of the limitations of logic-programming languages. As stated in Reference 12, through meta-level programming some inaccessible aspects of the underlying object-level system can be made explicit and treated as first-class objects.

An interesting subclass of meta-programming systems is constituted by reflective/introspective systems.<sup>13,14</sup> In this case, the system has a description of its structure and behaviour explicitly available.

As Reference 15 says, in order to make a system introspective 'the language-interpreter has to be able to construct an explicit model of a system and its current state at any moment in time. It is on the basis of these representations that the system will be able to introspect, i.e. to answer questions about itself and support actions upon itself'.

The number and variety of the questions and actions will depend on the power of the model, though of course there will always be program-representation issues beyond the reach of any practicable model.

If evaluated following this line of reasoning, a standard Prolog machine can be considered only a very poor reflective system. In fact, the model of the program that can be accessed and manipulated is represented only by the set of clauses composing it. The overall dynamic computational state of the Prolog machine is completely hidden. For this reason, when a more detailed model of the program is necessary, the most common technique used in the Prolog community is to design meta-interpreters<sup>8</sup> (i.e. interpreters for the language written in the language itself).

A very simple meta-interpreter for pure Prolog that forms the basis for building several extensions to the Prolog language can be written as follows:

```
solve(true).
```

```
solve((A, B):- solve(A), solve(B).
solve(A):- clause(A, B),solve(B).
```

It uses direct access to the internal representation of the program through the built-in predicate `clause/2`, and then reproduces the overall interpretation cycle at the Prolog level.

We argue that a less extreme solution can be adopted in some cases if the underlying machine supports a more detailed program model. This solution relies on avoiding (if possible) reproduction at meta-level of the overall behaviour of the interpreter. Some parts of the computational state of the underlying Prolog machine can be accessed directly and manipulated through reflection mechanisms<sup>16</sup> in this approach, which we shall describe below as *reflection* or *introspection*.

In order to design a reflective system three issues have to be taken into account:<sup>13-15</sup>

1. *A model of the object-program computation.* The representation of the model should be accessed and handled directly during meta-level computations. It is on the basis of this model that the system will be able to reason about itself and perform actions upon itself.
2. *A reflection mechanism* which triggers the computation from the (object-) level to the introspective (meta-)level domain (*upward reflection*) and vice versa (*downward reflection*).
3. *A causal connection* between the object-program model and system behaviour. Any modification in the model's representation at meta-level must cause a modification in the behaviour of the object-level system (*reflection*) and 'vice versa' (*reification*<sup>17</sup>).

One of the main problems in the definition of the object-program model is to choose the right level of abstraction for the reflective activities.

Since this work mainly concerns Prolog-database combinations, the object-program model has been tailored to this specific problem. In particular, we define the (object-level-model) as the triplet {CU ,CG,AUX}, where

- (a) the CU register represents the current object-level-unit
- (b) the CG register represents the current (sub-)goal
- (c) the AUX register represents auxiliary information as will be better explained in the next section.

Through this model, units, goals and other abstractions become first-class objects of meta-level computations, and can be used as arguments for procedures.

In CPU, object-programs and meta-programs can be separated into different but equally important units. The separation of object- and meta-programs introduces more structure into programs, which makes them more readable and easier to understand and modify.

Since object- and meta-level units have the same representation and computation capabilities, introspection can apply recursively.

Unit `mu` becomes the meta-unit of `u` when the predefined predicate `connect(mu)` is executed with success in `u`.

If this is the case, CPU provides an implicit (and default) upward reflection mechanism whenever a (sub-)goal is to be demonstrated in `u`. The CPU interpreter

makes the object-level model explicit and shifts up the meta-level code by trying to demonstrate the following goal in *mu*:

```
reflect_up((object-level-model))
```

There is one (meta-)variable argument of the `reflect_up` predicate, defined in *mu*, for each of the CU, CG and AUX registers. Each such variable ranges over the permitted values in the corresponding register.

If `reflect_up` succeeds, the resulting (object-level-model), in which some variables can be bound, is reflected in the object-level environment by an implicit downward reflection. If `reflect_up` fails, its failure is propagated to the relevant object-level computation.

During meta-level computations, a need to shift down (in order to force a particular <object-level > computation) may arise. An explicit downward reflection is forced by calling the predefined predicate:

```
reflect_down( < object-level-model > )
```

This predicate activates an object-level computation starting from the interpreter state specified in (object-level-model).

If this process terminates successfully, `reflect_down` ends with success too, and some variables in the meta-level environment can be bound by an implicit upward reflection. Otherwise `reflect_down` fails. In this way the causal-connection issue is solved without the need for a full meta-circular interpreter. Changes can be made directly to the basic interpreter by implicit upward and downward reflection mechanisms. Since any modification of the underlying machine state is accomplished in a disciplined way, consistency is guaranteed.

CPU does not need any predefine predicate to express communication between units: communication can be implemented by reflection mechanisms. For example, consider the following meta-rule in *mu*:

```
reflect_up(U,ask( UD,Goal),Aux):- reflect_down(UD, Goal,Aux)
```

Whenever the goal `ask(ud,g(a,X))` is called in *u*, the computation is reflected to the meta-level, and the right code found in *mu* is executed. During the ‘upward reflection’, the *U* variable is bound to the current CU register value (i.e. *u*), *UD* is bound to the called unit *ud* and *Goal* to `g(a,X)`. Reflect-down activates the proof of `g(a,X)` directly in the object-unit *ud* by forcing registers CU and CG to *ud* and `g(a,X)`, respectively. No change is performed on the AUX register. The correct execution of backtracking in *u* and *ud* is guaranteed by the underlying machine behaviour.

## REFLECTION TO EXPRESS PROLOG-DATABASE COMBINATION

By using reflection mechanisms, concepts usually frozen in language notations—or implemented in Prolog by using side-effects, built-in predicates (`assert` and `retract`) or by defining a full meta-circular extended interpreter—can be expressed simply by programming meta-units and connecting them dynamically with object-level units.

In the following, different forms of database combination will be expressed in CPU by using reflection mechanisms. In particular, context extension/contraction and inheritance mechanisms will be considered, since they can implement most of the basic concepts in structuring and combination of items of knowledge. Context extension/contraction is dynamic and expressed by operators associated with specific goals, while inheritance is usually static and associated with a specific unit.

### Contexts

In contextual logic programming<sup>18</sup> the logic program, statically viewed as a set of units, dynamically corresponds to the execution of goals in variable sets of units (contexts) representing the current line of reasoning.

Rather than evaluate a goal  $g$  in a particular unit  $u$  (denoted  $u \mid -g$ ) the user can dynamically choose to evaluate  $g$  in the union of different units (e.g.  $u_1 \cup u_2 \cup \dots \cup u_n \mid -g$ ).

More precisely, we define as a *context* an ordered list of unit names which varies during execution and determines the overall set of clauses to be considered in the proof of a goal.

In particular, if the current context is  $[U_{N^2}, U_{N-1}, \dots, U_1]$  the ordered set of clauses considered is

$$\begin{aligned} &\langle U_{N^2} \text{ clauses} \rangle \\ &\langle U_{N-1} \text{ clauses} \rangle \\ &\quad \vdots \\ &\langle U_1 \text{ clauses} \rangle \end{aligned}$$

Contexts can be considered as a very general and powerful mechanism for introducing modularity, hypothetical reasoning and viewpoints in logic programming. There are many ways to define and handle contexts.<sup>5,18-20</sup> For this reason, before freezing a particular policy of context handling, it would be very useful to experiment with the possible alternatives by using CPU and its reflection mechanisms. In CPU we represent contexts as dynamic ordered lists of units and treat these as first-class objects at meta-level where reflection is concerned.

The following meta-rule (referred to as a *context-rule* below) can be interpreted as a high-level specification of context implementation:

$$\begin{aligned} \text{reflect\_up}(\_, G, \text{Context}) :- & \text{member}(U, \text{Context}), \text{unit}(U), \\ & \text{reflect\_down}(U, G, \text{Context}), \end{aligned}$$

The third meta-rule argument (AUX register) represents the context in which the proof of the (sub-)goal  $G$  is to be tried. The rule body states the new actions performed at each resolution step in order to support contexts. An element  $U$  is extracted from the current Context list (`member/2`) and, if  $U$  is a unit (`unit/1`), the proof is performed within it (`reflect_down/3`).

The search space of a single unit is thoroughly explored by the underlying machine, whereas that of the overall context is explored by a generate-and-test strategy at meta-level.

### Context extension/contraction

In Prolog-based systems, operators extending contexts<sup>18</sup> cannot be commutative since the order of clauses in the text influences the proof of whatever is the current goal. For this reason, we introduce two operators to extend the current context with a new unit  $u$ :  $U \text{ adda } G$  and  $U \text{ addz } G$ . These predicates add  $U$  clauses at the top or at the bottom of the current context, respectively.

The implementation of these operators is straightforward by the use of CPU and reflection:

```
reflect_up(_, U adda G, C):- !, reflect_up(_,G, [U | C]).
reflect_up(_, U addz G, C):- !, append(C, [U], C1 ), reflect-up(_, G,C1 )
context-rule) .
```

When the  $\text{adda}/2$  operator is called, the new context is obtained by ‘stacking’  $U$  clauses on top of  $C$ . The  $\text{addz}/2$  operator is implemented in a similar way.

These operators allow us to avoid the use of non-logical Prolog predicates such as `assert` and `retract` to add or remove new clauses to the current unit. With  $\text{adda}$  and  $\text{addz}$  the underlying system guarantees automatic discarding of dynamically-added clauses at the end of each proof whether this has been successful or not.

Consider the following example (first reported in [Reference 19](#)): in the Prolog clause

```
g1:- g2(X),asserta(fact1 (X)), g3(Y),g4(Y),
    retract(fact1 (X)), g5(Z).
```

some problems arise when backtracking occurs. For example, if  $g3(Y)$  or  $g4(Y)$  execution causes backtracking on  $g2(X)$ , `fact1(X)` will be taken into account erroneously during  $g2(X)$  alternative execution. Moreover, if  $g5(Z)$  fails,  $g4(Y)$  and  $g3(Y)$  alternatives are tried without considering `fact1(X)`. The solution of these problems in Prolog can lead to a very cumbersome programming style.

By using contexts the same clause can be expressed as follows:

```
g1:- g2(X),create(U, [fact(X)]),
    U adda (g3(Y),g4(Y)),g5 (Z).
```

After successful  $g2(X)$  execution in context  $C$ , a new unit  $U$  is created dynamically. The new extended context, on which  $g3(Y)$  and  $g4(Y)$  are solved, is obtained by stacking  $U$  on the top of  $C$ . Context contraction is executed automatically when  $g3(Y)$  and  $g4(Y)$  are solved successfully or when backtracking occurs.

Since the above operators can be nested, they have been implemented by a recursive call to `reflect_up/3`. In this way, the goal  $u1 \text{ adda } (u2 \text{ addz } g1)$ , executed in the context  $[u0]$ , causes the proof of  $g1$  to be tried in the extended context  $[u1, u0, u2]$ .

### Context switch

The non-monotonic switch-context operator,  $C \rightarrow G$ , sets a new context,  $C$ , to solve goal  $G$ . Its meta-level implementation is

```
reflect_up(_, NewContext -> G):- !, reflect_up(_, G, NewContext).
context-rule).....
```

The current context is set to a new list (*NewContext*) on which the proof of the *G* goal is activated. At a lower level, context switching is achieved by setting the AUX register, i.e. by enforcing in it the value of the variable *NewContext*.

If the new context is simply a one-element list, the  $\rightarrow/2$  operator can be compared to an external call to a different unit. References to the called unit and the goal can be determined, and units created, dynamically. This kind of external call is thus suitable for sending messages to objects in logic-programming systems with object-oriented features. We note that if *NewContext* is an unbound variable, the meaning of the invocation *NewContext*  $\rightarrow$  *g* could be: ‘What is the context that can answer goal *g* successfully?’ This form of request can be considered an extended version of the intentional messages introduced in [Reference 21](#).

Operators on contexts of this kind can be used in hypothetical reasoning and to handle multiple viewpoints. We illustrate this by considering the following example, very similar to the one presented in [Reference 19](#) which deals with a simple form of hypothetical reasoning:

```
unit(u0).
on(a,b).
on(b,c).
on(c,table).
next(X,Y):- on(X,Y).
next(X,Y):- on(Y,X),
colour(b,black).
next_white(B):- next(B,X), colour(X,white),
```

The rule *next\_white* states when a block *B* is next to a block of colour white. Let us suppose that we know, in addition, that

```
colour(a,white) or colour(c,white).
```

The goal is to demonstrate that a block exists next to a white block. To solve this problem two alternative viewpoints (i.e. hypotheses) *v1* and *v2* are created:

```
unit(v1).
colour(a,white).

unit(v2).
colour(c,white).
```

Moreover, a *next\_white* rule has to be defined as follows to deal with the alternative viewpoints:

```
unit(main).
next_white(B):- [u0,v1] -> next_white(B),
                [u0,v2] -> next_white(B),
```

When the following goal is invoked:

```
?- main adda next_white(B)
```

the answers will be Yes, B=b.

Automatic context contraction and the switch operator avoids any interference between alternative viewpoints.

### Inheritance

Several object-oriented and frame-based systems introduce different inheritance policies in order to classify units and re-use knowledge. In Smalltalk<sup>22</sup> a simple inheritance is introduced; in Flavors<sup>23</sup> multiple inheritance and combination are supported; Actors<sup>24</sup> provide prototypes and delegation. In the Multilog<sup>25</sup> knowledge-based system, three different inheritance relations have been introduced: full inheritance, inheritance with exceptions and default inheritance.

Since it seems impossible to reach universal agreement on a single knowledge-sharing policy, reflection allows the programmer to express explicit meta-rules to define different policies. Inheritance policies allow the programmer to re-use implicitly the knowledge of a unit (u1) in order to demonstrate (sub-)goals in another unit (u2). Inheritance relations between units are expressed explicitly in CPU as facts of the following form:

```
super(u2,u1)
```

The inheritance relationship forms an acyclic directed graph of units on which inheritance policies can be defined.

By using backtracking, the ancestor/2 predicate:

```
ancestor(U,U)
ancestor(U,AU). - super(U, SU), ancestor(SU,AU)
```

finds all units belonging to a particular inheritance relationship, including the starting unit. The transitivity of the inheritance relation is expressed by the second clause of ancestor/2. Expressing super/2 and ancestor/2 explicitly as logic predicates leads to the possibility of using unification and deduction directly on them to define, for example, dynamic constraints on inheritance relationships.

Inheritance policies in logic programming can be conceived in different ways:<sup>21,26</sup>

1. If the success/failure (i.e. declarative) semantics is adopted, a clause is sought in the super-units of the current unit when it fails in that unit. Then the axiom set (context) associated with a goal in unit u is the union of the Horn clauses defined in u and in its super-units.
2. If the traditional call/return (i.e. procedural) semantics is kept, a clause is sought in the super-units of the current unit when it is not defined in that unit.

### Success/failure semantics

In CPU, the following meta-rule—referred to below as the *declarative-inheritance rule*—implements an inheritance policy with success/failure semantics. Communication between units is assumed to be performed by means of the ask predicate defined in the section on reflection, above.

```
reflect_up(_,G, Self):- ancestor(Self, U),
                        reflect_down(U, G, Self)
```

This rule states that the first place in which to look for the solution of G is the unit Self. This is the unit requested by ask to give the proof of G. If G fails in Self, the inheritance graph is traversed (*ancestor/2*).

The AUX register is used, in this case, to store the name of the unit in which the search has to be started (Self).

The role of AUX is assumed by the Self variable in Smalltalk and by the ‘Client’ variable<sup>27</sup> in delegation-based systems. In these systems complete search of the inheritance graph is guaranteed by the implementation layer.

With the previous meta-rule, a rough multiple inheritance is implemented automatically: if a unit has more than one super-unit the inheritance graph is searched, depth-first, using the backtracking mechanism of the standard Prolog interpreter. With suitable choices of goal arguments or units, this kind of approach can lead to quite sophisticated inheritance policies.

If the different units involved do not define disjoint sets of predicates, the previous meta-rule can find multiple solutions in multiple units (full inheritance). In this case, if negation is not involved, the union of axiom sets is monotonic since the axiom set of a unit is a superset of those of its super-units. However, in particular applications, only the solution found in the more specific unit has to be taken into account. A way to control the global search space should therefore be given. This is relevant for applications that introduce default and exception concepts. The problem of implementing non-monotonic knowledge-sharing policies is discussed in the next section.

### Contexts, inheritance and the cut operator

Context extensions and inheritance can be coupled by replacing context and inheritance rules with the following *context-inheritance* rule:

```
reflect_up(_, G, Context):- member(U,Context), ancestor(U,AU),
                            reflect_down(AU, G, Context).
```

All solutions are explored. The system first finds all solutions deriving from the inheritance relationship, then those deriving from the context list. The resulting logic programming system can cope with both static taxonomic relationships and dynamic combinations of knowledge in a uniform way.

When Prolog is extended with separate databases and knowledge-combination operators, the meaning of the ‘cut’ predicate has to be defined with care. There are at least two possible interpretations:

1. The cut has *global effects*. Its domain is, in this case, the complete set of units composing the current context and/or the inheritance graph.
2. The cut has only *local effects*. Its domain is statically confined to the unit in which it appears.

In CPU, we have followed the second interpretation because we believe that it is very difficult to write and understand programs where the cut has global effects that can only be determined dynamically. This choice does not limit the expressive power of the extended Prolog language.

Any global search policy in a context or in an inheritance graph can be expressed in terms of meta-rules. Cuts within meta-rules have a scope limited to the unit in which they appear, but simulate global effects on the dynamic, global object-level search space. For example, simple modifications of the declarative inheritance meta-rule of the previous section can implement different non-monotonic inheritance policies. The cut predicate can be used to avoid any search in the super-units. Depending on the position of the cut in the meta-rule body, overriding has different semantics. For instance, where just the first solution has to be produced (default inheritance), a cut must be added at the end of the body of the inheritance meta-rule.

Specific search strategies can be embedded in different CPU meta-units and properly selected, if needed, at system configuration time.

### Call/return semantics

If call/return semantics must be maintained, the *procedural-inheritance rule* can be defined as follows:

```
reflect_up(_, G, Self):-
  ancestor(Self, U),
  transform(G,G1),                /*generates from G, a new goal*/
  reflect_down(U, clause(G1, B), Self),!,    /*G1 with all arguments unbound*/
  reflect_down(U, G, Self), !.
```

This rule states that a unit *U* inherits each clause of its super-units, except if there is already in *U* a clause with head having the same predicate symbol and the same arity. Overriding occurs whether clause matching *G* is successful or not.

## COMPARISONS BETWEEN INTROSPECTION AND FULL META-INTERPRETATION

The comparisons reported here are an abridged and revised version of the comparisons presented in [Reference 28](#).

The case study selected to make comparisons between introspection and full meta-interpretation is context-handling, since it seems a simple and very powerful mechanism to express typical kind of abstractions that occur in artificial intelligence.<sup>19</sup>

If a full meta-interpretation technique is adopted to implement context handling, the Prolog machine's main loop and its computation rule must be reproduced. The AUX register is represented here as a meta-interpreter parameter.

A pure (i.e. not supporting built-in predicate) meta-interpreter capable of dealing with context-handling operators is sketched in [Figure 1](#).

The `clause/3` predicate allows us to find clauses defined in the unit specified as the first argument. The first argument of predicate `solve/2` represents the goal to be solved, whereas the second represents the current context where the goal has to be proved. The meta-interpreter above (called PMI, for pure meta-interpreter) is able to solve the switch-context and merge operators, as well as goal conjunctions. Note that no extra-logical Prolog predicates are supported in PMI. Writing a full meta-interpreter (called FMI) supporting standard Prolog (cut included) and context handling is not a trivial task: the FMI code becomes cumbersome and complex (see [Figure 2](#)). The first and second arguments of `solve/3` still represent the current goal and context. The third argument is needed to deal with the cut operator. FMI handles both context operators and the full set of built-in Prolog predicates.

In CPU, context handling can be expressed without the need for a full meta-interpreter, but simply by introspection (IN). As shown above, three meta-rules (written in unit `meta_context` and reported in [Figure 3](#)) are sufficient to implement context handling.

Each object-unit connected with `meta_context` will be able to support context handling. When the switch operator is called (rule `mr1.`), the new context is set to `NewContext`. When the `adda/2` operator is called (rule `mr2.`), the new context is obtained by 'stacking' `U1` clauses on top of `C`. Since the operators discussed above can be nested, they have been implemented with a recursive call to `reflect_up/3`. Rule `mr3.` can be interpreted as a sort of high-level specification of context implementation. The first meta-rule parameter is recorded in the stack for each resolution step, and represents the context in which proof of the (sub-)goal `G` of unit `U` is to be tried. Context implementation is performed by a simple version of 'generate-and-test', and backtracking on unit union is supported automatically by the underlying machine.

## COMPARISON OF THE TWO APPROACHES

With the help of CPU, the case study can easily be solved without the need for a full meta-interpreter. In fact, built-in Prolog predicates (cut included) are not reflected

```

solve(true,_) :- !.
solve((A,B),C) :- !,solve(A,C),solve(B,C).
solve((C -> G),_) :- !,solve(G,C).
solve(adda(U,G),C) :- !,solve(G,[U|C]).
solve(G,C) :- member(U,C),clause(U,G,B),solve(B,C).

member(X,[X,_]).
member(X,[_],T) :- member(X,T).

```

*Figure 1. A pure meta-interpreter for context handling (PMI)*

```

solve(G,C) :- solve(G,C,R), R==true.
solve(true,_,true) :- !.
solve(!,_,R) :- !,(R=true; R=Cut).
solve((A,B),C,R) :-
    !,solve(A,C,R1),
    ( R1==true, solve(B,C,R2),
      (R2==true, R=true;
       R2==cut, !, R=cut);
      R1==cut, !, R=cut).
solve((A;B),C,R):- !,
    ( solve(A,C,R1 ),
      (R1==true, R=true;
       R1==cut, !, R=cut);
      solve(B,C,R1),
      (R1==true, R=true;
       R1==cut, !, R=cut)).
solve(call(A),C,true):- !,solve(A,C,true).
solve(not(A),C,_) - solve(A,C,R),R=true,!,fail.
solve(not(A),C,true) - !.
solve(A,C,true) :- system(A),!,call(A).
solve((C -> G),_,R) :- !,solve(G,C,true).
solve(adda(U,G), C,R) :- !,solve(G,[UIC],true).
solve(G,C,R):- member(U,C), solve_local_cut(U,G,CR).

solve_local_cut(U,G,C,R):-
    clause(U,G,B),
    solve(B,C,R1),
    (R1==true, R=true;
     R1==cut, !, fail ),

member(X,[X,_]).
member(X,_,T) :- member(X,T).

```

Figure 2. A full meta-interpreter for context handling (FMI)

automatically at meta-level, but supported directly by the underlying machine.

Direct changes can be made to the run-time environment of the basic interpreter by upward and downward reflection mechanisms. For this reason the solution is more elegant, understandable and brief. The AUX register has proved very useful in implementing context handling. This register is very general: it can be bound to any Prolog term and handled only at meta-level, but its consistency (e.g. its correct treatment during backtracking) is guaranteed by the underlying machine.

Of course, despite the generality of the AUX register, the program model present in CPU is suitable only for a particular problem category (to which the case study belongs) but it becomes unsatisfactory if applied to different (more general) problems. In fact, there is no general optimal program model. For example, if one wants

```

unit(meta_context),

% switch rule
mr1. reflect_up(U, NewContext->G,OldContext:- !,
               reflect_up(U ,G, NewContext),

% merge rule
mr2. reflect_up(U,U1adda G, C):-!,
               reflect_up(U,G,[U1|C]).

% exec rule
mr3. reflect_up(U,G, Context):- member(U1 ,Context),
               refl_d_down(U1,G,Context).

member(X,[X,_]).
member(X,_,T) :- member(X,T).

```

Figure 3. Direct introspection to implement context bundling (IN)

to change the Prolog computation rule,<sup>29</sup> the program model of CPU is too superficial since the goal selection is left implicit. If this is the case, the more flexible and full meta-interpreter approach has to be followed to avoid direct intervention on the low-level code of the underlying machine.

### Performance comparisons

One of the main problems in using meta-programming techniques to build real systems is the overhead introduced by the presence of multiple computational levels. For this reason, the comparisons reported here focus on performance evaluations.

Performance comparisons have been made between the codes in Figure 1 (PMI), Figure 2 (FMI) and Figure 3 (IN). PMI and FMI run on a C-Prolog interpreter extended with modularity, whereas IN runs on CPU. The comparisons have been performed in two different cases: interpretation and partial evaluation.

The results have been normalized with respect to the IN solution. The test is based on the object-level program in Figure 4 and is performed with the goals

```

unit(u0).
nrev([],[]).
nrev([A | X],Y) :- nrev(X,Z),append(Z, [A], Y).

unit(u1).
append([],X,X).

unit(u2).
append([T | C],Y, [T | Z]) :- append(C,Y,Z).

```

Figure 4. The program on which performance comparisons have been made

?- solve([u0,u1, u2] -> nrev([1,2, . . . ..2 0],X).\_) for PMI and FMI,  
 ?- [u0, u1, u2] -> nrev([1,2, . . . ..2 0]X) for IN.

As will be clear in the following, the program above favours FMI performance results compared to those for IN and PMI. Cuts and built-in predicates are not used. Moreover, since there is not a great number of conjunctions in clause bodies, some overheads due to meta-interpretation are not made evident by the test.

### Interpretation

Following the interpretation approach, a modular Prolog has been implemented by extending a C-Prolog interpreter<sup>30</sup> running on a Sun 3/140 workstation. In particular, the C-Prolog clause representation has been modified to support separate Prolog databases, a new register (called CU) has been added and the unification algorithm has been partially modified. The value of the CU register is used at each resolution step to select the appropriate clause for unification.

At each resolution step, if the current unit is connected to a meta-unit, the interpreter makes the program model explicit and reflects it at meta-level. To implement the `reflect_down/3` built-in predicate, the interpreter has to force `reflect_down/3` arguments into the AUX, CU and CG (i.e. the register referring to the current goal) registers, respectively, and resume an object-level proof.

The test execution has given the results in [Table I](#).

We note that introspection is more efficient than meta-interpretation. In particular, the solution based on introspection is about twice as fast as that based on the pure meta-interpreter (PMI). This ratio notably increases if the full meta-interpreter (FMI) is taken into account. The overhead of meta-interpreter solutions is mainly due to the following reasons:

1. The computation rule is reproduced at meta-level. This overhead increases with the number of goal conjunctions in the clause body.
2. The explicit execution of the `clause/3` built-in predicate is slower than the equivalent operation in CPU due to the need to handle clauses as data structures.
3. A greater number of unifications is necessary to find the solution.
4. A greater number of shallow backtrackings<sup>11</sup> is performed since many alternatives for the `solve/3` predicate are present.

Moreover, system predicates in FMI are executed considerably more slowly than in IN. This overhead does not appear in the case study, since system predicates are not used in the program of [Figure 4](#).

Table I. Interpretation performance  
(results normalized with respect to IN)

IN	PMI	FMI
1	1.90	3.09

*Partial evaluation*

In the comparisons, partial evaluation techniques should not be ignored. In fact, partial evaluating<sup>31</sup> is considered as one of the most promising techniques to reduce the meta-programming overhead.

The fundamental idea of partial evaluation is to evaluate parts of the program which have enough input data for that purpose and keep unchanged those parts that do not allow such simplification. The basic mechanisms used in this process are

- (a) unfolding of procedure calls with their bodies
- (b) forward and backward propagation of data structures.

In logic-programming languages, unification and resolution automatically support these mechanisms. In logic-programming terms, partial evaluation can be described as follows:<sup>32</sup> given a program  $P$  and a goal  $G$ , partial evaluation produces a new program  $P'$ , which is  $P$  specialized for goal  $G$ . In other words,  $G$  and all its instances should have the same answers with respect to  $P$  and  $P'$ . The basic technique for obtaining  $P'$  from  $P$  is to construct partial search trees for  $G$  with respect to  $P$  and then extract  $P'$  from the definitions associated with the leaves of these trees.

The use of partial evaluation has a great relevance with respect to meta-level programming techniques. The application of partial evaluation to meta-programming has already helped extensively, for example, in defining the various layers of the Logix virtual machine<sup>8</sup> and in the design of a logic programming system that supports separate Prolog theories.<sup>33</sup> In all these systems, the partial evaluator is applied to a meta-interpreter with respect to the goal of an object program. The result is a specialization of the meta-interpreter for the program that can be seen as a 'compiled' version of the program incorporating all the additional features provided by the meta-interpreter. Since the 'compiled' version can be executed directly by the basic interpreter, it is much more efficient.

Since our tests refer to extended Prolog machines, a partial evaluator for standard Prolog has been modified accordingly to deal with modularity in the case of meta-interpreters and with modularity and reflection in CPU. The partial evaluators for PMI/FMI and IN have been obtained by modifying a partial evaluator for standard Prolog. This partial evaluator (called PE in the following) has been implemented as a Prolog meta-interpreter. Its main characteristics are:

1. Unfolding of a subgoal  $G$  is suspended when
  - (a)  $G$  is a built-in predicate with side-effect or insufficient instantiation of its arguments
  - (b)  $G$  is a recursive call (agreeing with a particular user-defined criterion)
  - (c)  $G$  can be solved with clauses whose bodies contain occurrences of cuts that cannot be eliminated.
2. When a subgoal is suspended, only forward data-structure propagation is performed.
3. Execution plans are produced for recursive calls by generating procedures with different names, one for each suspended goal  $G$ .
4. Further optimization have been performed by post-processing the resulting

On the basis of PE, two extended partial evaluators have been implemented to deal with modularity and reflection. To handle modularity, the only significant extension is to provide a means of handling the `clause/3` predicate.

In the case of IN, the extended partial evaluator is able to treat reflection mechanisms when a connection between object- and meta-level units is established. If this is the case, then for each object-level sub-goal the partial evaluator tries to unfold the corresponding `reflect_up/3` predicate in the meta-level unit. When the `reflect_down/3` predicate is reached, it tries to unfold the goal—specified as the third argument of `reflect_down/3`—with object-level clauses. Moreover, it has to maintain in a consistent way two variables simulating the AUX and CU registers.

The PMI and FMI solutions have been partially evaluated with respect to the program of [Figure 4](#) and the goal

```
?- solve([u0, u1 ,u2] -> nrev(Z,X),_).
```

The IN solution has been partially evaluated with respect to the program of [Figure 4](#) and the goal

```
?-[u0, u1 ,u2] -> nrev(Z,X),
```

The resulting optimized codes are reported in the [Appendix](#).

We expect efficiency to increase when partial evaluation is applied. [Table II](#) confirms this belief. This table reports the ratio between execution times of the original code and the partially-evaluated code for the case study under examination.

Execution of the test has given the results in [Table III](#) when normalized with respect to IN.

We note that introspection is still more efficient than full meta-interpretation, but the difference is significantly reduced. This is due to the fact that most of the full meta-interpretation overhead is eliminated, thanks to partial evaluation. For exam-

Table II. Partial evaluation vs. interpretation

IN	PMI	FMI
6.16	13.18	6.71

Table III. Partial evaluation test (results normalized with respect to IN)

IN	PMI	FMI
1	0.88	2.84

pie; the clause/3 predicate is no longer executed, the need for shallow backtracking is greatly reduced and many unifications are eliminated. The optimized codes of IN and PMI are similar in structure. Nevertheless, the IN version is a little slower than the PMI one. This is due to the fact that the IN `reflect_up/3` predicate has one more argument than the corresponding `solve/2` predicate in PMI. This argument represents the current unit (not significant in the case study but, of course, the partial evaluator cannot omit it).

### CONCLUSIONS

Since no agreement exists on knowledge-structuring policies, our design choice was to provide a single and fairly flexible method for implementing knowledge-sharing in logic programming. The resulting CPU system is based on reflection. Its main aim is to offer a flexible and extensible environment to test alternative knowledge-structuring policies.

CPU has been implemented on a Sun 3/140 workstation<sup>34</sup> to experiment with the features discussed above and has also been applied to several knowledge-representation issues, e.g. frames and viewpoints.

About ten experimental libraries of special reflective behaviors expressed in terms of meta-units have been defined in the CPU environment. The user can select a particular meta-unit implementing specific concepts and policies and extend the basic system by connecting them dynamically to different object-level units.

We believe that this approach helps to make logic programming more suitable for a number of artificial-intelligence applications requiring the use of alternative knowledge bases and frequent switching of contexts.

In particular, in the design and implementation of an advanced logic programming environment in the context of the ALPES Project (973) in the E.E. C. ESPRIT programme, reflection has been very useful for rapid prototyping and experiments with different and more complex policies of context handling,<sup>18,35</sup> than the ones presented here, without significant overheads.

A typical problem of meta-programming techniques is the overhead introduced by the multiple levels of interpretation. One advantage of the reflection mechanisms presented here is to reduce such an overhead by avoiding full meta-interpretation. Moreover, CPU programs that avoid use of reflection mechanisms do not run significantly slower than conventional Prolog programs. In fact, no overhead exists if reflection capabilities are not exploited (i.e. if units are not connected to meta-units).

As can be inferred from the performance results reported above, introspection allows a better trade-off between efficiency and extensibility. Only partial-evaluation techniques can make introspection and meta-interpretation comparable. In addition, introspection allows meta-programs to be more concise and readable.

### ACKNOWLEDGEMENTS

This work has been partially supported by the ALPES Project (973) in the E.E.C. ESPRIT programme and by the 'Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo' of the National Council of Research CNR under grant no. 890004269.

The work of the third author has been partially supported by ENIDATA S.p.A. We would also like to thank the anonymous referees and the editor in chief for their pertinent remarks and suggestions, and for the help they gave us in improving this paper.

## APPENDIX

**PMI partially-evaluated code with respect to the goal**

```
?- solve([u0, u1 ,u2] -> nrev(Z,X),_)

solve1 ([u0, u1 ,u2] -> nrev([], []),_).
solve1([u0, u1 ,u2] -> nrev([A | X | , [A],_):-
  solve5(nrev(X, []), [u0, u1, u2]).
solve1 ([u0,u1 ,u2] -> nrev([A | X],[C | Z] ),_):-
  solve5(nrev(X, [C | Y]), [u0, u1, u2]),
  solve11 (append(Y, [A], Z), [u0, u1 ,u2]).

solve5(nrev([], []), [u0, u1, u2]).
solve5(nrev([A| X], [A], [u0, u1, u2):-
  solve5(nrev(X, []), [u0, u1, u2]),
solve5(nrev[A|X], [C | Z]), [u0, u1, u2):-
  solve5(nrev(X, [C | Y]), [u0, u1, u2]),
  solve11 (append(Y, [A], Z), [u0, u1 ,u2]),

solve11 (append([], [A], [A]), [u0, u1 ,u2]).
solve11 (append([B | X],[A],[B | Z]),[u0, u1 ,u2):-
  solve11 (append(X, [A], Z), [u0, u1 ,u2]).
```

**FMI partially-evaluated code with respect to the goal**

```
?- solve([u0,u1 ,u2] -> nrev(X,Z),_)

solve1 ([u0,u1 ,u2] -> nrev([], []),_).
solve1 ([u0, u1 ,u2] -> nrev([_40 | 41 ],_39),_):-
  solve8(nrev(_41 ,_42),[u0 ,u1 ,u2],_43),
  ( _43== true,
    fold 16(_39, _40,_42);
    _43==cut,
    !,
    fail).

fold16([_40],_40,[]).
fold16([_44_45],_40, [_44 | 46):-
  solve17(append(_46, [_40],_45), [u0, u1, u2],_47),
  ( _47== true;
    _47 = =cut,
    !,
    fail),

solve8(nrev([], []), [u0, u1, u2],true),
solve8(nrev([_41 | _42],_39)[u0, u1 ,u2],_40):-
  solve8(nrev(_42, _43), [u0, u1 ,u21,_44),
  ( _44= =true,
    fold36(_39,_40, _41,_43);
    _44==cut,
    !,
    fail).
```

```

fold36([_41 ],true,_41 ,[]).
fold36([_45 | _46],_40,_41 ,[_45[_47]):-
  solve17(append(_47, [_41],_46), [u0, u1, u2],_48),
  ( _48== true,
    _40=true;
    _48==cut,
    |
    fail),
solve17(append( [], [_39],[_39]), [u0, u1, u2],true).
solve17(append([_42 | _43 ],[_39], [_42 | _44]), [u0,u1,u2],_41):-
  solve17(append(_43, [_39],_44), [u0, u1,u2],_45),
  ( _45== true,
    _41 =true;
    _45==cut,
    |
    fail
  ).

```

### IN partially-evaluated code with respect to the goal

```
?- [u0,u1 ,u2] -> nrev(Z,X).
```

```

[u0,u1 ,u2] -> nrev([],[]).
[u0,u1 ,u2] -> nrev([A,B],[B,A]).
[u0,u1 ,u2] -> nrev([A,B | L], R):-
  reflect_up1 ([u0, u1 ,u2], u0, nrev([B | L], LX)),
  reflect_up2([u0, u1,u2], u0,append(LX,[A], R)),
reflect_up1 ([u0, u1,u2], u0, nrev([], [])).
reflect_up1 ([u0, u1 ,u2], u0, nrev([A | L],X)):-
  reflect_up1 ([u0, u1 ,u2], u0, nrev(L, LX)),
  reflect_up2([u0, u1, u2], u0,append(LX, [A], X))
reflect_up2([u0, u1,u2],u0,append( [],[A], [A])).
reflect_up2([u0, u1 ,u2],u0,append( [B],[A], [B, A])).
reflect_up2([u0, u1, u2],u0,append([ B, C | X],[A], [B, C | Z])):-
  reflect_up3([u0, u1 ,u2], u2,append(X, [A],Z)).
reflect_up3([u0,u1,u2], u2,append([],[A], [A])).
reflect_up3([u0, u1,u2],u2,append( [B | X],[A],[B | Z])):-
  reflect_up3([u0,u1,u2],u2,append(X,[A],Z)).

```

### REFERENCES

1. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
2. K. Bowen, 'Mets-level programming and knowledge representation', *New Generation computing*, 3, (4), 359–383 (1985).
3. MProlog Language Reference Manual, Logicware Inc., Toronto (Canada), September 1985.
4. K. Nakashima, 'Knowledge representation in Prolog/KR', *International Symposium on Logic Programming*, Atlantic City (U.S.A.), IEEE Computer Society Press, 1984, pp. 126–130.
5. H. Bacha, 'Mets-level programming: a compiled approach in J.-L. Lassez (ed.), *Proceedings of*

- the 4th International Conference on Logic Programming*. Melbourne (Australia), The MIT Press, Cambridge, Massachusetts, 1987, pp. 394–409.
6. P. Mello and A. Natali, 'Programs as collections of communicating Prolog units', in B. Robinet and R. Wilhelm (eds), *Proceedings European Symposium on Programming (ESOP 86)*, Saarbruchen (WC) Lecture Notes on Computer Science no. 213, Springer-Verlag, Berlin (WG), 1986, pp. 274–288.
  7. K. Bowen and R. Kowalski, 'Amalgamating language and metalanguage in logic programming'. in K. L. Clark and S.-A. Tarnlund (eds), *Logic Programming*. Academic Press. London, 1982, pp. 153–172.
  8. S. Safra and E. Shapiro, 'Mets-interpreters for Real'. in H. G. Kugler (cd. ). *Information Processing 86*, Elsevier Science Publishers, 1986, pp. 271–278.
  9. P. H. Winston and B. K. P. Horn, *LISP*, second edn, Addison-Wesley, Reading. Massachusetts. 1984.
  10. T. Chikayama, 'ESP reference manual'. *ICOT Research Center Technical Report, TR-OW*. ICOT. Tokyo, February 1984.
  11. M. Bruynooghe. 'The memory management of Prolog implementation', In K. L. Clark and S.-A. Tarnlund (eds), *Logic Programming*. Academic Press. London (UK). 1982. pp. 83–98.
  12. J. des Rivieres. "Mets-level facilities in logic-based computational systems". *Preprints of the Workshop on Meta-Level Architectures and Reflection*. Cost-13 Project 'Advanced Issues in Knowledge Representation". Organizer. Alghero, September 1987.
  13. J. Batali, 'Computational introspection', *A.I. Memo no. 701*. MIT Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts. February 1983.
  14. B. C. Smith, 'Reflection and semantics in LISP", *Report No. CSLI-88-8*. Stanford University Center for the Study of Language and Information, Stanford University, Palo Alto, July 1981.
  15. P. Maes, 'Introspection in knowledge representation', in B. du Boulay, D. Hogg and L. Steels (eds), *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI 86)*, Brighton. July 1986, North-Holland. Amsterdam. 1986, pp. 256–269.
  16. R. Weyhrauch. 'Prolegomena to a theory of mechanized formal reasoning'. *Artificial Intelligence Journal*, **13**, (1), 133–170 (1980).
  17. D. Friedman and M. Wand. 'Reification: reflection without meta-phy~ic'. *ACM Conference on LISP and Functional Programming*. ACM. Austin. August 1984. pp. 348–355.
  18. A. Porto and L. Monteiro. 'Contextual logic programming'. in G. Levi and M. Martelli (eds). *Proceedings of the 6th International Conference on Logic Programming*, Lisbon. The MIT Press, Cambridge. Massachusetts, 1989. pp. 284–299.
  19. M. Cavafieri, E. Lamma and P. Mello. 'An extended Prolog machine for dynamic context handling. in Y. Kodratoff (ed.), *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI 88)*. Munich, August 1988. Pitman Publishing. London (UK). 1988. pp. 284–289.
  20. E. Lamma, P. Mello and A. Natali, 'The design of an abstract machine for efficient implementation of contexts in logic programming', in G. Levi and M. Martelli (eds). *Proceedings of the 6th International Conference on Logic Programming*. Lisbon. The MIT Press. Cambridge. Massachusetts. 1989, pp. 303–317.
  21. H. Gallaire, 'Merging objects and Logic programming: relational semantics.' *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*. Philadelphia (PA). August 1986, Morgan Kaufmann Publishers. Los Altos (CA). 1986. pp. 754–758.
  22. A. Goldberg and D. Robson. *Smalltalk -80, The Language and its Implementation*. Addison Wesley. Reading. Massachusetts. 1983.
  23. D. A. Moon, 'Object-oriented programming with flavors'. in N. Myerowitz (ed.). *Proceedings of the ACM Conference on Object-Oriented Programming, Languages and Applications (OOPSLA'86)*. Special Issue of *SIGPLAN Notices*. **21**, (11). (1986).
  24. R. E. Filman and D. P. Friedman. *Coordinated Computing*. McGraw-Hill. New York. 1984. chapter 11.
  25. H. Kauffman and A. Grumbach. 'MULTILOG: MULTIPLE worlds in LOGic Programming'. in B. du Boulay, D. Hogg and L. Steels (eds). *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI 86)*. Brighton. July 1986. North-Holland. Amsterdam. 1986, pp. 291–305
  26. L. Leonardi and P. Mello. 'Combining logic- and object-oriented paradigms'. *Proceedings of the 21st Annual Hawaii International Conference on System Sciences (HICSS-21)*. The Computer Society of the IEEE. 1988. pp. 376–385.

27. H. Lieberman, 'Delegation and inheritance: two mechanisms for sharing knowledge in object-oriented systems', *Actes des Journees Afcet-Informatique Langues Orientes Objet, BIGRE+ GLOBULE*, No. 48, January 1986.
28. M. Cavalieri, E. Lamma, P. Mello and A. Natali, 'Meta-programming through direct introspection: a comparison with meta-interpretation techniques', in J. W. Lloyd (ed.), *Proceedings Workshop on Mets-programming in Logic Programming (META88)*, Bristol, June 1988, pp. 293-305; also in H. Abramson and M. H. Rogers (eds), *Mets-Programming in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989, chapter 21, pp. 399-415.
29. J. W. Lloyd, *Foundations of Logic Programming*, Second Edn, Springer-Verlag, Berlin, 1987.
30. F. Pereira (ed.), *C-Prolog User's Manual*, SRI International, Menlo Park, CA, February 1987.
31. R. Venken, 'A prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimization', in T. O'Shea (ed.), *Proceedings of the 6th European Conference on Artificial Intelligence (ECA 184)*, Piss, September 1984, Elsevier Science Publisher B. V., Amsterdam, 1984, pp. 91-100.
32. J. W. Lloyd and J. C. Shepherdson, 'Partial evaluation in logic programming', *Technical Report CS-87-90*, Department of Computer Science, University of Bristol, Bristol, December 1987.
33. G. P. Coscia, P. Franceschi, G. Levi, G. Sardu and L. Terre, 'Mets-level definition and compilation of inference engines in the epsilon logic programming environment', in R. A. Kowalski and K. A. Bowen (eds), *Proceedings of the 5th International Conference on Logic Programming*, Seattle, The MIT Press, Cambridge, Massachusetts, 1988, pp. 359-373.
34. 'ALPES Final Report', *ALPES (ESPRIT Project 973)*, September 1989.
35. P. Mello, 'Inheritance as combination of Horn clause theories', *Pre- Proceedings Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, February 1989, pp. 255-271; also to appear in M. Lenzerini, D. Nardi and M. Simi (eds), *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Wiley, Chichester.