logic theories. Section 5 is devoted to the implementation in LML of two examples, aiming at giving the flavour of how to use the language in typical A.I. applications. Some conclusions are discussed in Section 6.

2. The functional kernel

The functional language which provides the meta-level of LML is essentially the kernel of Standard ML [17] equipped with non-strict semantics. The type system of the language includes a set of predefined types — e.g. int, string, → (function space) — and the possibility of defining new types via discriminating union, cartesian product and recursion. The following are examples of data type definitions:

```
type bool = true | false
type nat = zero | succ of nat
```

Functions are defined in a declarative style by means of pattern matching over formal parameters. For example:

```
val    Plus (zero,n) =    n  |
       Plus (succ n,m) = succ(Plus(n,m))
val    IfThenElse (true,x,y) = x  |
       IfThenElse (false,x,y) = y
```

The type of any expression can be statically inferred and it can contain type variables (for polymorphic functions). In the previous example, the following types are inferred:

```
Plus: nat × nat → nat
IfThenElse : bool × α × α → α
```

where α is a type variable.

New polymorphic types can be introduced by the user, as the following type "α list", equipped with the constant "[]" (empty list) and the infix cons "::":

```
type α list = []  |  ::  of  α × α list
```

Functions can be higher order, as the following example of the classic Map function shows:

```
val    Map f [] = []    |
       Map f (a :: R) = (f a :: Map f R)
with   Map : ( α → β ) → ( α list → β list )
```

Finally, functions are non strict, i.e. they can deal, in general, with undefined values or partially undefined data structures. For instance, given the following definition of the partial predecessor function:

```
val    Pred (succ n) = n
```

the evaluation of:

```
IfThenElse(true, zero, Pred zero)
```

yields zero even if (pred zero) is undefined, since the second argument is not needed in such an evaluation. Since type constructors are non-strict functions too, it is possible to define and use partially defined and potentially infinite data structures. The following definitions are examples of this feature:


```
val   Zeroes = zero :: Zeroes
val   From n = n :: From (succ n)
val   Numbers = From zero
```

As a consequence, it is possible to construct functions which work on infinite data without necessarily diverging. In the example:

```
val    First zero L = []  |
       First succ n (a::R) = (a :: First n R)
val    FirstTwenty = First 20
```

where 20 stands for $succ^{20}$(zero).
Then the (lazy) evaluation of

```
FirstTwenty  Numbers
```

results into the list of the first 20 natural numbers, although Numbers denotes an infinite list.

Non-strictness is a considerable expressive enhancement for a functional language, since it captures in a natural way some programming aspects like interactivity, input/output and memory sensitivity, which strict functional programming languages are not able to deal with, without compromising referential transparency [4,9,20].

3. Theories in LML

The logic programming paradigm is embedded into the functional kernel of LML through a new data type, the data type of logic theories. Basically, a logic theory is a set of rules, which are extended clauses allowing the use of negation in clause bodies. Logic theories share the same data type defined in the functional component, in the sense that only constant and constructors symbols introduced through type declarations can be used to build terms appearing in the rules.

Once defined, a logic theory is a denotable value, which can be bound to an identifier in the environment, like in the following relational re-definition of the + operation over natural numbers, which uses the type definition nat of the previous section:

```
val Peano = { plus(zero,x,x).
              plus(succ x, y, succ z) :- plus(x,y,z).}
```

As another example, consider the following enumeration type Node, introducing labels for the nodes of a graph:

```
val  Node = a | b | c | ... | z
```

An actual graph may be depicted through the following theory declaration, made by unit clauses only:

```
val  Graph = { edge(a,c).
               edge(b,c).
               edge(c,f).
               ...         }
```

The following theory defines the transitive/reflexive closure of a graph by means of a predicate called reachable:

```
val Closure = {reachable(x,x).
               reachable(x,y) :- edge(x,z),reachable(z,y).}
```

4. The Use of Logic Theories

Once we know how to define theories, some other mechanisms are needed in order to:
- put them together to form more complex theories and
- put them at work to get results.

The mechanism provided for the second purpose is called *set-expression*, while the *intensional operators* are introduced to cope with the first issue.

4.1 Intensional Operators

A major feature of LML is the ability of handling logic theories as ordinary, denotable data: the intensional operators are the means for gluing theories together and obtaining more complex ones as a result. The word intensional is intended to stress the point that these operators manipulate theory definitions as a whole — they do not perform pointwise manipulations of the corresponding extensions. Three basic intensional operators are provided: union, intersection and negation, together with a renaming operator:

```
rename Id1,…,Idk by Id'1,…,Id'k in Theory
```

which allows one to change predicate names within a theory, in order to avoid name clashes or undesired bonds. Let us discuss union and intersection first.

4.1.1 Union and Intersection

- **union** (P,Q) is a theory-valued expression which denotes the theory obtained by putting the clauses of theories P and Q together.

- **intersection** (P,Q) is a theory-valued expression which denotes the theory obtained from P and Q in the following way.

If p(t1,...,tn):-Body1. is a clause of P and
p(u1,...,un):-Body2. is a clause of Q and
there exists ϑ = mgu ((t1,...,tn),(u1,...,un))
then
p((t1,...,tn)ϑ) :- (Body1)ϑ, (Body2)ϑ is a clause of **intersection** (P,Q).

Union and intersection are two complementary means of putting two separate theories together. Querying the theory union (P,Q) means letting theories P and Q cooperate during the query evaluation process, since each theory can exploit rules and facts of the other one in order to draw conclusions. As an instance, the set-expression:

```
{ x | reachable(a,x) wrt union(Closure,Graph) }
```

which refers to the theories Closure and Graph introduced before, denotes the set of the nodes of the graph modelled by the theory Graph which are reachable from node a according to the reachable relation defined in the theory Closure. Notice that the latter refers to a relation edge which is not defined in it: thus, Closure may be viewed as a kind of parametric module, and this usage of the union operator as a way of instantiating it with an actual edge relation.

In a dual way, querying the theory intersection(P,Q) means constraining theories P and Q to agree at each step of the query evaluation process, since the theory intersection(P,Q) draws conclusions obeying to the conjunction of rules and facts of both P and Q. To exemplify this point, let us complicate a little the Graph/Closure example. Suppose the type Node represents towns in a given country:

```
type Node = town of string
```


Thus, town("Frisco") and town("LA") are values of type Node. Suppose now that a theory RoadAtlas includes a relation highway, defined in the following way: there is a fact highway(X,Y) if town X and town Y are linked through highways.

```
val RoadAtlas =
{…
 highway(town("Frisco"),town("LA")).
 … }
```

The Closure theory can be adapted to the new setting with a minor change:

```
val Closure = { reachable(x,x).
                reachable(x,y) :- highway(x,z),reachable(z,y).}
```

Now, suppose we are interested in knowing all the towns that may be reached from a given town, with the constraint that each town must have at least 100,000 inhabitants (data about population are stored as facts of the relation inhabitants in the theory Pop). The following theory Closure1 provides a suitable re-definition of the "reachable" predicate for that purpose:

```
val Closure1 =
        intersection( {reachable(town(x),town(y)) :-
                                    inhabitants(y,n), n≥100,000.},
                      Closure).
```

In the above definition, the single-clause theory which is going to be intersected with the Closure theory expresses the fact that "going from a town to another is worthwhile only if the destination is a large town". The predicate name reachable is used in order to allow the correct matching between clauses during theory intersection. So it is possible to achieve the desired result in the following way:

```
{x|reachable(town("Frisco"),x)
 wrt union(union(Closure1,RoadAtlas),Pop)}
```

In fact, due to the definition of the intersection operator, the actual definition of reachable in Closure1 is:

```
reachable(town(x),town(x))  :-  inhabitants(x,n), n≥100,000.
reachable(town(x),town(y))  :-  highway(town(x),z),
                                reachable(z,town(y)),
                             inhabitants(y,n), n≥100,000.
```

4.1.2 Negation

In this section the theory-level negation operator is discussed. In LML negation is dealt with by a particular approach to constructive negation in logic programming— called intensional negation in [1,2,14] — as opposed to the usual approach of negation as failure [7,12]. The basic idea is that a negative literal like ~p(t) is viewed not as the unary logical connective "~" applied to the atom p(t), but instead as a particular kind of positive atom, with the special predicate name "~p". According to this principle, a suitable definition of the "~p" predicate must be provided, which allows to evaluate queries involving it. If one is able to provide such a definition of "~p", then negative information can be computed in the same way ordinary, positive information is computed, thus avoiding the asymmetry of the negation-as-failure rule. Fortunately, it is possible to derive systematically the clauses defining predicate "~p" from those defining its positive counterpart "p". For instance, consider the following type representing formulas of the propositional calculus, together with a theory introducing some basic deduction rules:

```
type Prop = atom  of  string      |
            and   of  Prop × Prop |
            or    of  Prop × Prop |
            not   of  Prop
```

```
val Deduction = { thm(atom A)    :- axiom(A).
                  thm(and(P,Q))  :- thm(P),thm(Q).
                  thm(or(P,Q))   :- thm(P).
                  thm(or(P,Q))   :- thm(Q).
                  thm(not P)     :- ~thm(P).   }
```

Actually, the theory Deduction is completed by the system with a hidden component which defines the ~thm relation. In the example, such a definition is:

```
~thm(atom A)    :- ~axiom(A).
~thm(and(P,Q))  :- ~thm(P).
~thm(and(P,Q))  :- ~thm(Q).
~thm(or(P,Q))   :- ~thm(P),~thm(Q).
~thm(not P)     :- thm(P).
```

The way such a transformation is carried out is completely described in [1,2,14]. What is important to notice here is that the occurrence of the literal ~thm(P) in the positive component of the theory Deduction is nothing but a "call" to the predicate ~thm defined in the hidden, negative component, while conversely the occurrence of the literal thm(P) in the negative component is a call to the original predicate in the positive component.

Now, theory-level negation is defined as follows: given a theory P, the negated theory not P is obtained by interchanging the definition of the positive predicates with that of the negative predicates in the hidden component. In the example, the definition of thm and ~thm in the theory not Deduction are like that of ~thm and thm in theory Deduction, respectively. As a consequence, being a theorem in not Deduction is the same as being a non-theorem in Deduction.

4.2 Set Expressions

Set-expressions are the means for denoting the set of values obtained by evaluating a query with respect to a specified theory. The general syntax for set-expressions is the following:

```
{ Vars | L1,…,Lk wrt Theory }
```

where the conjunction of literals L1,…,Lk is a query, Vars is a tuple of variables, and each variable occurring in Vars must occur also in L1,…,Lk. As an example, let us refer back to the Peano theory. The set-expression:

```
{ (x,y) | plus(x,y,3) wrt Peano }
```

denotes the set of pairs

```
{(0,3), (1,2), (2,1) (3,0)}
```

The overall design of the language relies on a non-strict semantics, and coherently also set-expressions are lazily evaluated, in the sense that the extension of a set is never computed unless explicitly needed or requested. This is accomplished through a set of *extensional* operators which are listed in the sequel. SE stands for a generic set-expression and BE a generic boolean expression.

- `for each x in SE do E`
  results in the set of values obtained by evaluating the expression E for each element of SE.
- `all x in SE such that BE`
  evaluates to the set of elements of SE satisfying the boolean expression BE.
- `get x in SE with BE`
  evaluates to the unique element of SE satisfying BE, if any, otherwise it yields a failure.
- `v isin SE`
  evaluates to true if v is an element of SE.

- `exists x in SE such that BE`
  evaluates to true if BE holds for some element of SE.
- `forall x in SE holds BE`
  evaluates to true if BE holds for all elements of SE.
- `iterate f over SE with x0`
  evaluates to `f(…f(x0, x1),x2,…,xn)` where `{x1,…,xn}` is any enumeration of the extension of SE: it is equivalent to the *fold* (left) operation over lists. The result is not affected by the elements ordering providing that f satisfies:
  `f(f(x,y),z) = f(f(x,z), y)`
- `choose x in SE`
  selects non-deterministically an element of SE.

5. Examples

The aim of this section is to point out the expressive power of LML through two examples. The first one is an implementation of the Master Mind game, and its purpose is to show how the functional and logic components of LML can be exploited to address the conceptually different aspects of the problem by using the appropriate programming paradigm.
The second example is concerned with the definition in LML of a number of operators on theories implementing different inheritance relations. The purpose of this example is to point out the expressive power of the operators on theories and their suitability for a rational reconstruction of knowledge representation mechanisms.

5.1 Master Mind

This example is inspired from [21] and deals with guessing the secret code in the game of master mind. The game goes as follows: player A chooses a secret code, a list of N distinct decimal digits (usually N equals 4 for beginners, 5 for advanced players). Player B makes guesses, and queries player A for the number of *bulls* (the number of digits which appear in identical positions in the guess and in the code) and *cows* (the number of digits which appear both in the guess and in the code, but in different positions). The code is determined when a guess has N bulls.

First of all, let us define what a code is through the following type declarations:

```
type digit = 0|1|...|9
type code = digit list.
```

With the above definition a code is an arbitrary list of digits: notice that the type code is defined instantiating the polymorphic type α list of section 2. To get actual codes, e.g. of length four and with different digits, we can define the following theory:

```
theory Is_A_Code =
   { isacode([d1, d2, d3, d4]) :- isadigit(d1),
                                  isadigit(d2),
                                  isadigit(d3),
                                  isadigit(d4),
                                  d1≠d2≠d3≠d4.
     isadigit(0).
         .
         .
     isadigit(9). }
```

where [d1, d2, d3, d4] is used as a shorthand for d1 :: d2 :: d3 :: d4 :: [].

The whole game is coded through the recursive function MasterMind which, given a secret code and the codes already guessed, identifies a reasonable code to try next. The declarative knowledge of the problem is coded in a logic theory, named Bulls_Cows, which contains the definitions of two main relations, bulls and cows.
bulls(C1, C2, n) (resp. cows(C1, C2, n)) holds if C1 and C2 contain n digits in the same (resp. possibly different) positions. The relation agree is also defined, such that agree(C1, C2, b, c) holds if the number of bulls and cows, given C1 as the secret code and C2 as the guess, are b and c respectively. The purpose of the agree relation is to identify a code C2 as a reasonable guess, given that a previous guess C1 has produced b bulls and c cows, only if C1 agrees with C2 on the number of bulls and cows.

```
theory Bulls_Cows =
{bulls([], [], zero).
bulls((H::T),(H::T'), succ N) :- bulls(T,T',N).
bulls((H::T),(H'::T'), N) :-  H ≠ H',
                              bulls(T, T',N).

cows((H::T),L,succ N) :- member(H,L),
                         cows(T,L,N).
cows((H::T),L, N) :- ~member(H,L),
                        cows(T,L,N).

member(H,(H::T)).
member(X,(H::T)) :- X ≠ H, member(X,T).

agree(C1,C2,b,c) :- C1≠C2,
                    bulls(C1, C2, b),
                    cows(C1, C2, c).}
```

Given the previous definition of the theories Bull_Cows and Is_A_Code, the main function MasterMind is then defined as follows:

```
fun MasterMind SecretCode Trials =
    let  val
      Reasonable code Guessed_Codes =
           let  val
                A = {c1 | guessed(c1, b, c), ~agree(code, c1, b, c)
                          wrt union(Guessed_Codes, Bulls_Cows)}
           in   if isempty A then true else false;
      ToTry = all x in {y | isacode(y) wrt Is_A_Code}
               such that Reasonable x Trials
    in
     let  val
           NewCode = choose ToTry;
           (NewB, NewC) = get x in {(b,c) |   bulls(SecretCode,NewCode, b),
                                               cows(SecretCode, NewCode, c)
                                               wrt Bulls_Cows } with true
     in
           if NewB < 4
           then
             MasterMind SecretCode union(Trials, {guessed(NewCode,NewB,NewC)})
           else
             (NewCode, Trials)
```

The function Reasonable is a boolean-valued function which determines whether a given new code is consistent with the set of codes already guessed. These are assumed to be represented in the theory Guessed_Codes by means of assertions of the kind
guessed(t, b, c) meaning that code t has been already guessed and it has produced b bulls and c cows as a result . A code is in fact reasonable if it is consistent with all the codes already guessed: this is in turn determined through the relation agree of the theory Bull_Cows. The current NewCode is then computed choosing a code from the set ToTry of all the possible codes which are consistent with the

codes already guessed. The function MasterMind calls itself recursively if the NewCode produces less than 4 bulls, with the second parameter updated in order to record the last guessed code and the number of bulls and cows corresponding to it. Otherwise the correct code together with all the guesses are returned as the final result. Of course the first call to the function MasterMind should be:

```
MasterMind SC empty
```

where SC is the actual secret code and the initial theory representing guesses is empty.

5.2. Hierarchies

Many ways of structuring knowledge according to some hierarchical specification can be found in the knowledge representation literature. Semantic networks and frames [6,19] are only some of the best known examples.
Intuitively speaking, a hierarchical relation between two theories states that the information, i.e. the predicate definitions in our framework, present in a SUPER theory is inherited by, i.e. becomes visible to, a SUB theory which is hierarchically linked to the former. A trivial solution can be that the information included in a SUPER theory is fully inherited by its sub-theory. More sophisticated forms of inheritance can be specified in order to express a sort of exception mechanism by stating that a SUB theory inherits from a SUPER theory everything but what is re-defined in itself.

In the following, given two theories T_i and T_j, $preds_{ij}$ will stand for the set of predicates which are defined in both theories. T_i and T_j can be hierarchically linked by means of two kinds of theory valued operators:

- **is_a**(T_i, T_j) : T_i is the SUB theory and T_j is the SUPER theory. The resulting theory contains all the predicate definitions of T_i and inherits the definitions of predicates occurring in T_j only. Notice that the definitions in T_j of predicates in $preds_{ij}$ are replaced by the corresponding definitions in T_i.

- **constraint**(T_i, T_j) : T_i is the SUB theory and T_j is the SUPER theory. The resulting theory contains the intersection of the definitions of predicates in $preds_{ij}$ and inherits the definitions of predicates occurring in T_j only.

The previous hierarchical operators can be defined in terms of the basic meta-level operators of LML: in the following definitions we use the notation $T\uparrow preds$ to denote the clauses of the theory T defining predicates not belonging to the set of predicate symbols preds.

Definition *(Is_a Operator)*
Given two theories T_1 and T_2 :

$$\text{is_a}(T_1,T_2) = \textbf{union}(T_1, T_2\uparrow preds_{12}) \qquad \blacklozenge$$

Definition *(Constraint Operator)*
Given two theories T_1 and T_2 :

$$\text{constraint}(T_1,T_2) = \textbf{union}\ (\textbf{intersection}(T_1, T_2),T_2\uparrow preds_{12}) \qquad \blacklozenge$$

The operators of LML have been characterized in [16] as defining an algebra over a class of logic theories. Typical algebraic properties for the operators have been proved and their characterization in terms of success and finite failure sets has been discussed. This algebraic framework can be used for proving properties of new operators defined in terms of the basic ones. Below a number of properties of is_a and constraint which can be proved in this fashion is shown. Full definitions and proofs can be found in [5].

Proposition *(Properties of is_a and constraint)*
Let OP∈ {is_a, constraint} and T_1,T_2,T_3 be theories. Then:

i) $OP(T_1,T_1) = T_1$
ii) $OP(T_1,T_2) \neq OP(T_2,T_1)$
iii) $OP(OP(T_1,T_2),T_3) = OP(T_1, OP(T_2,T_3))$ ♦

Note that commutativity does not hold, as expected, is_a and constraint being hierarchical operators. The previous operators can be also studied in terms of the success and finite failure sets, denoted by SS and FF respectively.

Proposition
Let T_1,T_2 be theories. Then:

i) $SS(is_a(T_1,T_2)) \supseteq SS(T_1)$
ii) $FF(is_a(T_1,T_2)) \subseteq FF(T_1)$
iii) $SS(constraint(T_1,T_2)) \subseteq SS(T_2)$
iv) $FF(constraint(T_1,T_2)) \supseteq FF(T_2)$ ♦

Finally, let us compare the two hierarchical operators described so far. The relation between them is quite evident and is stated by the following proposition.

Proposition
Let T_1,T_2 be theories. Then:

i) $SS(constraint(T_1,T_2)) \subseteq SS(is_a(T_1,T_2))$
ii) $FF(constraint(T_1,T_2)) \supseteq FF(is_a(T_1,T_2))$ ♦

Multiple hierarchical systems, where a SUB theory is hierarchically linked to several SUPER theories, can also be described through the basic LML operators. Roughly, given a multi-hierarchy (see Figure 1):

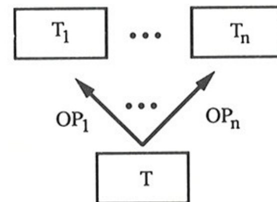

Figure 1

where $OP_i \in \{is_a, constraint\}$, the intuitive meaning is that the resulting theory is the composition of the knowledge separately inherited by the SUB theory from all the SUPER theories.
In particular, if some of the SUPER theories contain different definitions of the same properties then these definitions should not conflict each other, but instead they should contribute separately to the resulting knowledge.
The above consideration leads to the following definition of multi-hierarchical operator. Let multi-link(T_0, <T_1, OP_1>, ... , <T_n, OP_n>) denote a multi-hierarchy as the one depicted in Figure 1.

Definition *(Multi-link)*
Let $T_0, T_1, \ldots, T_n$ be theories. Then:

multi-link(T_0, <T_1, OP_1>, ... , <T_n, OP_n>) = **union**($OP_1(T_0,T_1),\ldots,OP_n(T_0,T_n)$)

where union($x_1,\ldots,x_n$) is a shorthand for **union**(x_1,**union**(..., **union**(x_{n-1},x_n)...)). ♦

In the case when all the links are is_a links and $preds_{ij}=\{\}$ for each i,j=1,...,n (i≠j), the multi-link in Figure 1 can be viewed as (see Figure 2):

Figure 2

where {j1,...jn} is a permutation of {1,...n}. As a consequence, under this restriction, the multi-link operator can be defined as a concatenation of is_a links.

Proposition
Given n theories $T_1,\ldots,T_n$ such that $preds_{ij}=\{\}$ for each i,j=1,...,n (i≠j) and given a theory T_0:

multi-link(T_0,<T_1,is_a>,..,<T_n,is_a>) = is_a(T_0, is_a(T_{j1}, is_a(..,is_a(T_{jn-1},T_{jn})...))

where {j1,...jn} is a permutation of {1,...n}. ♦

6. Conclusions

In this paper the distinguishing features of the language LML have been presented, along with two examples of LML programming, both taken from typical, though different, A.I. applications.

Many interesting aspects of the definition of LML have been omitted in this paper, due to the lack of space. Among others:

- operational issues concerned with the query evaluation process for set-expressions as well as its integration to the reduction-based evaluation of the functional component;
- semantics issues involving both the complete model-theoretic semantics of the logic component and the overall denotational semantics of the language;
- the extensions to the basic polymorphic type checking scheme of ML needed for the treatment of logic theories and theory-valued operators.

The interested reader is referred to [16,3,18] for details on the above mentioned issues.

Currently we are investigating extensions to the language involving further operators on logic theories, aimed at providing LML with the full power of a meta-language for logic programming. For instance, further mechanisms are needed in order to manipulate logic theories at the clause level, so that meta-interpreters can be coded in the language. These mechanisms should be designed in the same spirit of the existing meta-level operators, that is they should be based on well understood semantics grounds [13]. On the other hand, the suitability of the language for representing other reasoning mechanisms (e.g. default reasoning, abductive reasoning) are under investigation.

References

[1] Barbuti, R., Mancarella, P., Pedreschi, D. and Turini, F. "Intensional Negation of Logic Programs: examples and implementation techniques", in *Proc. TAPSOFT '87, LNCS 250*, 96–110 (1987).

[2] Barbuti, R., Mancarella, P., Pedreschi, D. and Turini, F. "A Transformational Approach to Negation in Logic Programming", *to appear in Journal of Logic Programming* (1989).

[3] Bertolino,B., Mancarella,P., Meo,L., Nini,L. ,Pedreschi,D.and Turini,F. "A Progrees Report on the LML Project". In*Proc. of the International Conf. FGCS 89*, ICOT, 675–684, (1989).

[4] Bird, R. and Wadler, P. *Introduction to Functional Programming*, Prentice Hall (1988).

[5] Brogi, A., Mancarella, P., Pedreschi, D. and Turini, F. "Hierarchies through Basic Meta-level Operators", to appear in *Proceedings of Meta-90, 2nd Workshop on Meta-programming in Logic*, Leuven, Belgium (April 1990).

[6] Charniak, E. and McDermott, D. *Introduction to Artificial Intelligence*, Addison Wesley (1985).

[7] Clark, K.L. "Negation as Failure", in *H.Gallaire and J.Minker (eds.), Logic and Data Bases*, Plenum Press, New York, 292–322, (1978).

[8] Gallaire, H. "Boosting Logic Programming", in: Proc. Fourth Int. Conf. on Logic Programming, Melbourne, Australia 962-988 (1987).

[9] Henderson, P. *Functional Programming: Application and Implementation*, Prentice Hall (1980).

[10] Kowalski, R.A. *Logic for Problem Solving*, Elsevier North Holland, New York (1979).
[11] Kowalski, R.A. "Logic Programming", in: Proc. IFIP'83 133–145 (1983).
[12] Lloyd, J.W., *Foundations of Logic Programming*, Springer Symbolic Computation Series, Berlin, (1987).
[13] Lloyd, J.W., "Directions for Meta-Programming" in*Proceedings of the International Conference on Fifth Generation Compure Systems,* Tokyo, (1988).
[14] Mancarella, P. *Intensional Negation of Logic Programs*. Ph.D. Thesis, University of Pisa (in Italian) (1988).
[15] Mancarella,P., Pedreschi,D. and Turini,F. "Functional Metalevel for Logic Programming", in D.Nardi and P.Maes (eds.), *Meta-Level Architectures and Reflections*, 329–344 (1988).
[16] Mancarella,P. and D. Pedreschi. "An Algebra of Logic Programs", in *Proc. of Fifth International Conference, Symposium of Logic Programming*, Seattle, 1006-1023 (1988).
[17] Milner, R. "A proposal for Standard ML", in *Proc. of 1984 ACM Symp. on LISP and Functional Programming* 184–197 (1984).
[18] Pedreschi,D. *Logic Programming: Compositional Semantics, Algebraic Structures and Program Completions*.Ph.D. Thesis, University of Pisa (in Italian) (1988).
[19] Rich,E. *Artificial Intelligence,* McGraw-Hill Int. Book Company, (1983).
[20] Richards, H. "The pragmatics of SASL for programming applications", *Technical Report ARC 82-15*, Austin Research Center, Borroughs Corporation (1982).
[21] Sterling,L. and Shapiro E., *The Art of Prolog*, the MIT Press (1986).

Estensioni di ordine superiore a Prolog sono necessarie

Stefania **Costantini** Pierangelo **Dell'Acqua** Gaetano Aurelio **Lanzarone**

Universita' degli Studi di Milano
Dipartimento di Scienze dell' Informazione
Via Moretto da Brescia 9, I–20133, Milano (+39–2–7575.222)
e–mail costanti@imiucca.bitnet
lanzaron@imiucca.bitnet

Sommario

Nella comunita' della programmazione logica sembra essersi creata di fatto una dicotomia tra chi ritiene che Prolog sostanzialmente abbia gia' tutti i costrutti espressivi necessari ad affrontare nella pratica applicazioni anche complesse e sofisticate, e chi, ritenendone alcuni insodisfacenti sotto qualche aspetto, mira invece a definire nuovi linguaggi volti a razionalizzare o estendere il Prolog. In questo lavoro si sostiene la seconda posizione, e si propone un linguaggio di programmazione logica con caratteristiche metalinguistiche superiori a quelle di Prolog dal punto di vista sia espressivo che inferenziale. Si discutono in particolare i motivi per cui le capacita' di Prolog di esprimere e utilizzare relazioni di ordine superiore, ovvero di trattare relazioni come dati, sono insoddisfacenti, e si mostra come questi aspetti vengono affrontati nel linguaggio proposto, esemplificandone l'utilita'.

1. Introduzione

Warren [Wa82] ha per primo accreditato la tesi secondo cui Prolog ha gia' la capacita' di trattare funzioni e relazioni come 'first class data objects', e che pertanto l'introduzione di costrutti linguistici piu' espressivi in questa direzione non potrebbe aggiungere niente di sostanziale. Questa tesi e' stata poi ripresa da altri, o esplicitamente, ad esempio in [SS86] cap. 17, o implicitamente, come nei libri, proliferati negli ultimi tre anni, che introducono all'utilizzo di Prolog nello sviluppo di sistemi basati sulla conoscenza.

Mutuandoli dalle piu' usate funzioni di ordine superiore dei linguaggi funzionali (Lisp), Warren considera a supporto della sua tesi alcuni esempi, tra cui il seguente.

La relazione hanno_proprieta(L,P) e' intesa vera se ogni elemento della lista L ha la proprieta' espressa dal predicato unario P. Poiche' P e' una variabile che sta per un simbolo di predicato, hanno_proprieta(L,P) e' una relazione di secondo ordine. Usando direttamente una variabile predicativa, la sua definizione sarebbe:

```
hanno_proprieta([T|C],P):-P(T),hanno_proprieta(C,P).
hanno_proprieta([],_).
```

Warren argomenta che e' inutile complicare la sintassi per usare variabili predicative, in quanto Prolog dispone degli strumenti necessari per simularle. Infatti, la definizione precedente puo' essere espressa nella forma:

```
hanno_proprieta([T|C],P):-applica(P,T),hanno_proprieta(C,P).
hanno_proprieta([],_).
applica(p1,X):-p1(X).
.....................
applica(pn,X):-pn(X).
```

dove p1,...,pn sono tutti i simboli di predicato corrispondenti alle proprieta' che si vogliono considerare (piu' in generale, si possono definire clausole analoghe per predicati di qualunque arieta'). In [SS86] si aggiunge che, se si vuole evitare di elencarle, l'insieme delle clausole applica puo' essere sostituito da un'unica clausola facendo ricorso ai predicati predefiniti univ e call:

```
applica(P,L):-G=..[P,L],call(G).
```

Warren conclude che la traduzione da variabili predicative a loro simulazione con applica riporta le prime nella logica del primo ordine standard; la forma puo' essere scomoda, ma comunque eventuali estensioni di Prolog per trattare i predicati come dati non renderebbero fattibile niente che non lo sia gia' in Prolog, e in questo senso sarebbero unicamente zucchero sintattico, che e' notoriamente questione di gusti.

In verita', la conclusione di Warren appare almeno affrettata, in quanto la simulazione proposta (o qualunque altra soluzione possibile in Prolog) presenta problemi molto seri.

Il fatto di trattare un simbolo ora come termine, ora come predicato come avviene in applica, ovvero di trasformare con univ un termine in una meta, eseguibile con call, non puo' essere considerato attinente alla logica del primo ordine standard, bensi' e' basato su una caratteristica meramente implementativa di Prolog, come e' bene espresso in [MO84]: "The definition of 'call' is based on the fact that most Prolog implementations use the same set of symbols for predicates and functions (this causes no syntactic ambiguity) and thus a Term has a naturally corresponding Atom. Hence 'call' is defined to be that predicate such that call(X) is equivalent to Y where Y is the Atom corresponding to the term X. Thus 'call' provides a method for evaluating a Term which has been constructed in a program and is accordingly related to 'EVAL' in Lisp".

Mycroft e O'Keefe concludono che la difficolta' (nel loro contesto, dell'introduzione in Prolog dei tipi) "arises from the conflation of object- and meta-levels in one language... A satisfactory resolution to this problem waits on the introduction of an explicit metalevel or the construction of a genuinely reflective Prolog".

Sterling e Shapiro stessi commentano che la soluzione da loro ripresa dall'articolo di Warren e' provvisoria, in attesa di migliori sviluppi: "It is possible that the active ongoing work on both extending the logic programming model with higher-order constructs, and integrating it with functional programming, will change the picture" ([SS86] p. 282).

Il fatto che i costrutti metalinguistici di Prolog (a cui si deve tanta parte della potenza espressiva di Prolog indispensabile nelle applicazioni) non hanno una semantica logica, e pertanto non l'hanno tutti i programmi che ne fanno uso (metaprogrammi, di cui i metainterpreti sono parte), e' stata piu' recentemente oggetto di una dettagliata analisi teorica da parte di Hill e Lloyd, che giungono ad una conclusione analoga: "These so-called 'impure' aspects of Prolog cause many practical Prolog programs to have no declarative semantics at all and to have unnecessarily complicated procedural semantics". "It is clear that Prolog's meta-programming problems can be traced to the fact that it doesn't handle the representation requirements properly" ... "most currently available Prolog systems do not make a clear distinction between the object level and the meta-level, do not provide explicit language facilities for representation of object level expressions at the meta-level". L'obiettivo e' quindi "an investigation of appropriate language facilities for meta-programming and the application of meta-programming techniques" ([HL88] pp. 27 e 39).

Naturale risvolto dei problemi semantici sopra indicati e' l'intero ordine di problemi che attengono alle possibilita' effettive di Prolog nel trattare i predicati come dati, e quindi alle reali capacita' espressive di Prolog in questo campo cruciale per la rappresentazione di conoscenza e di metaconoscenza. Nel seguito discuteremo perche' la simulazione proposta da Warren e' limitativa, e illustreremo esemplificandoli alcuni aspetti relativi a relazioni di ordine

superiore che non sono trattabili in Prolog.

In sintesi, la causa di tutti i problemi sopra menzionati e' che le caratteristiche metalinguistiche di Prolog sono un'approssimazione molto rozza di un metalinguaggio. Nei paragrafi seguenti presentiamo un linguaggio, chiamato Reflective Prolog (RP), che realizza un'approssimazione migliore relativamente agli aspetti considerati. Nel paragrafo 2 si introducono i concetti su cui Reflective Prolog e' basato e si riassumono le principali caratteristiche del linguaggio. Nel paragrafo 3 si riprende il problema di rappresentare relazioni di ordine superiore mostrando come viene affrontato in Reflective Prolog. Nel paragrafo 4 si forniscono elementi di comparazione con altre proposte presenti in letteratura e si traggono alcune conclusioni. La presentazione e' prevalentemente informale ed esemplificativa. Un'esposizione piu' completa e formale del linguaggio e della sua semantica si trovano in [CL89b], [Co89].

2. Reflective Prolog

Il principale requisito per un linguaggio che consenta di esprimere e usare metaconoscenza e' la capacita' di formalizzare la conoscenza e le regole di inferenza su livelli diversi ma connessi e comunicanti. Cio' richiede essenzialmente un dispositivo (detto di "naming" o autoriferimento), con cui ad un certo livello si possa fare riferimento alle espressioni linguistiche del livello sottostante, e un dispositivo (detto di riflessione) per passare da un livello all'altro, in entrambe le direzioni (si veda [ACS86] per una presentazione della problematica).

Reflective Prolog e' un linguaggio logico di programmazione e di rappresentazione della conoscenza dotato di queste capacita'. Esso consente di esprimere proposizioni sia di livello oggetto che di uno o piu' metalivelli, e di definire al livello superiore regole di inferenza per i livelli inferiori, aggiuntive rispetto a quella standard del suo interprete.

L'alfabeto di un programma Reflective Prolog ha, in piu' rispetto a quello di un programma Prolog [Ll87], due tipi di *metacostanti*: quelle *quotate* (scritte fra doppi apici, ad es. "f") e quelle *parentesizzate* (scritte fra parentesi angolari, ad es. <p>). Le prime sono intese come nomi per le costanti, i simboli di funzione e le variabili; le seconde come nomi per i simboli di predicato. L'alfabeto contiene inoltre due tipi di metavariabili, le metavariabili *predicato*, scritte con primo carattere '#', e le metavariabili *funzione*, scritte con primo carattere '$'. Vi sono infine alcuni simboli riservati, in particolare i simboli di predicato solve, theory_solve, theory_fact (unari) e name (binario), e i simboli di funzione predication, predicate, function, functor, arity, args.

I simboli di funzione riservati servono per costruire una nuova classe di termini, i *termini nominali*, che sono di tipo *funzione* o di tipo *relazione*, e fungono da nomi di termini e di formule atomiche, rispettivamente. Ad esempio, il nome del termine f(a) e' il termine:
function(functor("f"),arity(1),args(["a"])),
e il nome dell'atomo q(f(a),X) e' il termine:
predication(predicate(<q>),arity(2),
    args([function(functor("f"),arity(1),args(["a"])),"X"])).

Nell'utilizzo effettivo del linguaggio si puo' usare una forma abbreviata per i nomi (gestita dal parser del linguaggio) ottenuta omettendo i simboli di funzione riservati e l'arieta'. In particolare, i due nomi precedenti diventano rispettivamente "f"("a") e <q>("f"("a"),"X"). E' possibile anche esprimere nomi di nomi (e nomi di nomi di nomi, ecc.). Ad esempio, il nome del nome di f(a), cioe' il nome di "f"("a"), e' ""f""(""a"").

Nel seguito, il nome di un termine o atomo A e' indicato con ↑A; il passaggio da A a ↑A e' detto *referenziazione*, il passaggio inverso *dereferenziazione*. In Reflective Prolog vi e' il predicato binario predefinito (con interpretazione prefissata) name(α,A), vero se α = ↑A, dove il primo argomento deve essere un termine nominale ed il secondo un termine. Proceduralmente, gli argomenti devono essere o termini ground o variabili non istanziate (ma non termini parzialmente istanziati). Se entrambi gli argomenti sono istanziati, name(α,A) riesce se ↑A = α; se solo il primo argomento e' istanziato, esso viene dereferenziato ed assegnato al secondo; se solo il secondo argomento e' istanziato, esso viene referenziato ed assegnato al primo; se i due argomenti sono entrambi non istanziati, name fallisce.

I termini si possono considerare suddivisi in *termini oggetto*, che corrispondono a quelli del Prolog, e *metatermini*, che contengono almeno una metacostante, o un termine nominale, o una metavariabile come sottotermine. Gli atomi di metalivello (di contro a quelli oggetto) contengono almeno un metatermine fra gli argomenti. Queste definizioni consentono termini e atomi i cui componenti sono a livelli di referenziazione differenti, come ad esempio in pred("f"("a"),<p>,g(""b"",c),h(X)).

Le metavariabili denotano i metatermini (con relative estensioni dell'algoritmo di unificazione): le metavariabili predicato spaziano sulle metacostanti parentesizzate (vengono cioe' legate a nomi di predicato) mentre le metavariabili funzione spaziano sui metatermini in genere. Ad esempio, $X puo' assumere come valore "c", "V", "fun", fun("c",V),

<pred>("a","X"), mentre #P puo' assumere come valore <pred>.

I programmi Reflective Prolog sono scritti in clausole di Horn, con alcune restrizioni. Se la testa di una clausola e' un atomo oggetto, anche gli atomi della coda devono essere atomi oggetto (in tal caso la clausola e' detta *clausola oggetto*). Se la testa di una clausola e' un atomo di metalivello, gli atomi della coda possono essere sia atomi oggetto che atomi di metalivello (*clausola di metalivello*). Questa distinzione corrisponde ad un punto di vista secondo cui a ciascun livello e' accessibile la conoscenza rappresentata ai livelli inferiori, ma non viceversa. Se il predicato della testa e' il predicato distinto solve (*clausola solve*), il suo argomento deve essere un termine nominale di tipo relazione (ossia il nome di una meta). Il predicato solve puo' apparire nelle clausole che definiscono solve stesso o uno dei suoi predicati ausiliari. L'insieme di tali clausole e' detto *livello di metavalutazione* del programma; l'insieme delle rimanenti clausole (oggetto e/o meta) e' detto *livello base*. I predicati di livello base possono essere usati al livello di metavalutazione, ma non viceversa (le ragioni verranno discusse in seguito). Infine, un predicato non puo' essere sia usato che menzionato nella stessa clausola: se in una clausola compare l'atomo p(...), allora non puo' comparire la metacostante <p>.

Il seguente e' un esempio di programma in Reflective Prolog:

```
            /*  livello di metavalutazione  */
1       solve(#P($X,$Y)):- simmetrica(#P),solve(#P($Y,$X)).
2       solve(#P($X,$Y)):- equivalente(#P,#Q),solve(#Q($X,$Y)).

            /*  livello base  */
3       friend(raf,gianni).
4       simmetrica(<friend>).
5       equivalente(<amico>,<friend>).
```

Il livello base contiene la rappresentazione di diversi tipi di conoscenza. Il fatto 3 esprime conoscenza oggetto, cioe' dichiara che la relazione friend vale tra gli individui raf e gianni. Il fatto 4 esprime metaconoscenza, poiche' dichiara simmetrica la relazione friend (ed e' quindi un metafatto). Infine, il metafatto 5 dichiara che le relazioni amico e friend sono equivalenti.

Il livello di metavalutazione definisce strategie di inferenza ausiliarie (rispetto a quella standard dell'interprete), espresse dalle regole 1 e 2. La prima definisce la simmetria di una relazione binaria: gli oggetti denotati da $X e $Y sono in relazione secondo #P se la relazione denotata da #P e' stata dichiarata simmetrica e gli oggetti denotati da $Y e $X sono in relazione secondo #P. La seconda regola esprime l'equivalenza di due relazioni binarie: gli

oggetti denotati da $X e $Y sono in relazione secondo #P se la relazione denotata da #P e' stata dichiarata equivalente a quella denotata da #Q e gli oggetti denotati da $X e $Y sono in relazione secondo #Q.

Nelle clausole di metavalutazione e' possibile utilizzare i predicati distinti unari theory_fact e theory_solve che, come solve, sono definiti sui termini nominali di tipo relazione. In particolare, theory_fact(↑A) e' vero se A e' un fatto (clausola unitaria), mentre theory_solve(↑A) e' vero se A e' dimostrabile al livello base (cioe' senza fare uso delle clausole di metavalutazione). A differenza di solve, il significato di theory_fact e theory_solve e' predefinito, nel senso che essi non ammettono clausole definitorie. L'uso di theory_solve aiuta a migliorare l'efficienza in quei casi in cui la non applicabilita' delle regole di metavalutazione sia evidente in anticipo.

La regola di inferenza dell'interprete di Reflective Prolog e', come in Prolog, la risoluzione mediante unificazione, con strategia da sinistra a destra, in profondita', e ritorno indietro cronologico. La risoluzione e' stata pero' estesa (rispetto a quella del Prolog) per realizzare l'integrazione fra i vari livelli, considerando le clausole solve come regole di inferenza ausiliarie. Il meccanismo di integrazione e' la *riflessione*, che consente di cambiare dinamicamente il livello al quale si svolge una dimostrazione. Il Reflective Prolog si basa in particolare sulla riflessione *implicita*, nel senso che le condizioni per i passaggi di livello non devono essere indicate esplicitamente nel programma, bensi' sono integrate nella risoluzione, nel modo seguente.

Una sottometa Am (se non e' theory_solve(...) o theory_fact(...)) puo' essere risolta o, come in Prolog, mediante una clausola la cui conclusione A unifica con Am (questo caso include le chiamate ricorsive di solve), oppure, in aggiunta (se non e' solve(...)), mediante una clausola di metavalutazione con conclusione solve(α), se il suo nome ↑Am unifica con α. Quest'ultimo e' un caso di riflessione (implicita) verso l'alto: la nuova meta contiene infatti le condizioni della clausola di metavalutazione; le sottomete solve(...) o theory_solve(...) sono escluse, perche' non e' possibile meta-metavalutazione. Viceversa, una sottometa solve(M) o theory_solve(M) puo' essere risolta mediante una clausola base con conclusione A se M e ↑A unificano. In quest'ultimo caso avviene una riflessione verso il basso: la nuova meta contiene le condizioni della clausola base. Infine, una sottometa theory_fact(M) puo' essere risolta mediante una clausola unitaria base A se M e ↑A unificano.

La procedura di risoluzione adottata dall'interprete Reflective Prolog e' tale che ciascuna

sottometa (anche solve(...)) viene tentata innanzitutto al livello base e poi, in caso di fallimento, al livello di metavalutazione. In particolare, ogniqualvolta una meta fallisce a livello base, si sposta l'elaborazione a livello di metavalutazione mediante l'invocazione del predicato solve con argomento il nome della meta fallita. Questo determina l'attuazione delle strategie alternative di deduzione espresse dalle clausole definitorie di solve, in quanto la risoluzione estesa si comporta in modo equivalente ad entrambi i livelli. L'invocazione ricorsiva di solve in tali clausole ha l'effetto di tentare il suo argomento in prima istanza (dereferenziato) al livello base, quindi, se non riesce, ancora al livello di metavalutazione (dando effettivamente seguito alla ricorsione).

Ritornando all'esempio precedente, il quesito:

```
?- amico(gianni,X).
```

viene dimostrato nel modo seguente. Il fallimento a livello base sposta l'elaborazione a livello di metavalutazione, attuando la referenziazione automatica della meta fallita (con cui si ottiene <amico>("gianni","X")) e la sua unificazione con l'argomento di solve (che istanzia la metavariabile #P alla metacostante <amico>, la metavariabile $X alla metacostante "gianni", e la metavariabile $Y alla metacostante "X"). Al livello di metavalutazione si utilizza dapprima la regola 1, che esprime la simmetria. Non essendoci pero' un metafatto che dichiara la relazione amico simmetrica, si passa a tentare la regola 2, dove la ricerca di una relazione equivalente ad amico riesce, istanziando la metavariabile #Q a <friend>.

Mediante la chiamata ricorsiva a solve, il suo argomento <friend>("gianni","X") viene prima dereferenziato e tentato direttamente a livello base (dove fallisce), quindi riconsiderato al livello di metavalutazione mediante la regola della simmetria; essendo la relazione friend dichiarata simmetrica, si ottiene la meta <friend>("X","gianni") eseguita quindi con successo (mediante la chiamata ricorsiva di solve) al livello base. Cio' determina il soddisfacimento del quesito iniziale, con istanziazione della variabile X a raf.

Si puo' ora motivare la distinzione fra livello base e livello di metavalutazione, e la proibizione di utilizzare al livello base i predicati di metavalutazione. Quello che si vuole in effetti proibire e' l'uso (diretto o indiretto) di solve al livello base, che corrisponde a *riflessione esplicita*, in quanto specifica un passaggio al livello di metavalutazione. Oltre che contraria alla filosofia del Reflective Prolog (secondo cui le regole solve hanno natura dichiarativa, ed e' demandato alla risoluzione sfruttarne la valenza procedurale) una chiamata del genere e' anche inutile: una regola come p(X):-solve(<q>("Y")) e' del tutto equivalente a

p(X):-q(Y), visto che, se fallisce al livello base, la sottometa q(Y) viene automaticamente metavalutata.

La semantica del Reflective Prolog [Co89] e' definita in modo sostanzialmente simile a quella del linguaggio delle clausole di Horn [Ll87], con le dovute estensioni; di conseguenza, i programmi Reflective Prolog godono del corrispettivo delle proprieta' semantiche peculiari dei programmi logici. Poiche' le regole solve sono intese come un'estensione della semantica dei predicati di livello base sui quali si applicano, e' stata introdotta una nuova classe di modelli, i *modelli riflessivi*, nei quali questa estensione viene resa effettiva. Ciascun programma ha un modello minimo riflessivo di Herbrand (rispetto al quale la risoluzione estesa e' corretta e completa), che e' in generale piu' ampio del modello minimo tradizionale, in modo da includere quelle conseguenze che derivano dal considerare le regole di metavalutazione come regole di inferenza ausiliarie.

3. Relazioni di ordine superiore in Reflective Prolog

Riprendiamo ora il tema, sollevato nell'introduzione, della simulazione di variabili predicative con il predicato applica, nelle due formulazioni ivi esposte. La seconda soluzione (un'unica clausola per applica) non consente di usare il predicato hanno_proprieta con il secondo argomento non istanziato (per trovare quali proprieta' sono soddisfatte dagli elementi della lista in ingresso), perche' il secondo argomento di univ in applica deve essere istanziato se il primo non lo e'. La prima soluzione (piu' clausole applica) e' piu' laboriosa, ma consente tale possibilita'. Le due formulazioni hanno quindi caratteristiche complementari, e non e' possibile in Prolog una soluzione allo stesso tempo compatta e flessibile. In Reflective Prolog si hanno senza difficolta' entrambe le caratteristiche:

```
hanno_proprieta([T|C],#P):-applica(#P,T),hanno_proprieta(C,#P). hanno_proprieta([],_).
solve(<applica>($P,$A)):-name($P,#P),solve(#P($A)).
```

Ad esempio:

```
p(a).
q(b).
p(c).
hanno_proprieta([a,b],<q>).   /* no */
hanno_proprieta([a,c],#P).    /* #P = <p> */
```

Al di la' dell'esempio di Warren, si puo' considerare un'ampia varieta' di casi in cui e' utile 'reificare' relazioni, in modi non riconducibili all'utilizzo di applica.

Consideriamo il problema di passare da una proprieta' espressa con un predicato unario alla rappresentazione mediante una relazione binaria, e viceversa, ad esempio uomo(X) e e_un(X,uomo). Con formulazione analoga a quella di Warren si avrebbe:

```
uomo(X):-e_un(X,uomo).
e_un(X,uomo):-applica(uomo,X).
applica(uomo,X):-uomo(X).
```

Da notare che il passaggio in un senso e' possibile mediante un'unica clausola (la prima) senza la chiamata di applica; nell'altro senso no, almeno se si vuole consentire l'uso con il secondo argomento libero, cioe' le ultime due clausole non possono in questo caso essere sostituite dall'unica clausola:

```
e_un(X,uomo):-uomo(X).
```

Tuttavia la definizione risulta circolare; infatti va in ciclo infinito per ogni quesito istanziato che non trova un fatto corrispondente (oppure se nel programma i fatti sono scritti dopo le regole). La soluzione in Reflective Prolog e' piu' generale, ed inoltre non e' soggetta a cicli infiniti:

```
solve(<e_un>($X,$Y)):-name($Y,#P),theory_solve(#P($X)).
solve(#P($X)):-name($X,Y),e_un(Y,#P).
```

Dati ad esempio:

```
uomo(ugo).
e_un(roberto,<uomo>).
```

si ottiene risposta positiva ai quesiti:

```
?-e_un(ugo,<uomo>).
?-uomo(roberto).
```

risposta negativa (senza loop) ai quesiti:

```
?-uomo(gianni).
?-e_un(gianni,<uomo>).
```

e risposta #P = <uomo> al quesito:

```
?-e_un(ugo,#P).
```

Ancora, si puo' notare che si possono usare relazioni di inclusione tra classi, in una qualsiasi delle due seguenti forme:


```
e_un(<uomo>,<umano>).
e_un(X,<animale>):-e_un(X,<umano>).
```

e combinarle con le rappresentazioni precedenti premettendo alle due clausole solve gia' indicate la clausola:

```
solve(<e_un>($X,$Y)):-theory_fact(<e_un>($Z,$Y)),solve(<e_un>($X,$Z)).
```

Si ottiene cosi' ad esempio risposta positiva ai quesiti:

```
?-umano(ugo).
?-e_un(ugo,<animale>).
```

Questa combinazione non e' possibile in Prolog. Piu' in generale, la simulazione del comportamento della solve (e delle combinazioni delle sue clausole di definizione) non e' possibile in Prolog, neanche con metainterpreti (questo aspetto e' mostrato piu' in dettaglio in [CL89a]).

Un'altra classe di relazioni di secondo ordine e' relativa all'esprimere una proprieta' di una relazione, come la simmetria usata nel paragrafo 2. Come ulteriore esempio, supponiamo di avere i seguenti fatti:

```
a_nord_di(milano,roma).
a_sud_di(messina,napoli).
```

Naturalmente e' opportuno rappresentare il fatto che queste due relazioni sono una inversa dell'altra. Questo non e' esprimibile in clausole di Horn in entrambe le direzioni:

```
a_nord_di(X,Y):-a_sud_di(Y,X).
a_sud_di(X,Y):-a_nord_di(Y,X).
```

perche' ovviamente queste regole possono portare a cicli infiniti. La soluzione in Reflective Prolog e' immediata e naturale:

```
inversa(<a_nord_di>,<a_sud_di>).
simmetrica(<inversa>).
solve(#P($X,$Y)):-inversa(#P,#Q),solve(#Q($Y,$X)).
solve(#P($X,$Y)):-simmetrica(#P),solve(#P($Y,$X)).
```

Essa consente ad esempio di derivare a_sud_di(roma,milano).

Anche estendendo leggermente l'esempio di Warren si pongono in Prolog problemi di ciclo infinito se, con i fatti riportati prima, si vuole ad esempio aggiungere un'equivalenza tra p e q in modo da derivare sia hanno_proprieta([a,b], p) che hanno_proprieta([a,b],q). In Prolog occorre aggiungere le clausole:

```
equivalente(p,q).
equivalente(q,p).
applica(P,X):-equivalente(P,Q),applica(Q,X).
```

Ora i fatti sopra menzionati sono derivabili, pero' ogni quesito non dimostrabile, come:

```
?-hanno_proprieta([d],p).
?-hanno_proprieta([d],q).
```

causa un ciclo infinito pur non essendoci nella definizione una circolarita' evidente, come era invece il caso dell'esempio sulla trasformazione di proprieta' in relazioni. Questo non avviene nella soluzione in RP, che consiste nell'aggiungere le regole 1 e 2 dell'esempio al § 2 ed i fatti:

```
equivalente(<p>,<q>).
simmetrica(<equivalente>).
```

Gli esempi discussi sopra, benche' molto semplici, sono sufficientemente basilari e rappresentativi di quell'ampia classe di problemi, non trattabili in Prolog, che richiedono appunto di esprimere e utilizzare relazioni di ordine superiore. Tali esempi hanno il proposito di illustrare, oltre che il linguaggio proposto, un aspetto sufficientemente ricorrente e generale da dare un'idea complessiva di alcune possibilita' di utilizzo di Reflective Prolog. Questo aspetto riguarda le modalita' di rappresentazione della conoscenza.

E' noto che la particolare rappresentazione adottata influenza notevolmente l'utilizzabilita' della conoscenza di un dato sistema basato su di essa. La scelta della rappresentazione e' un a priori dello sviluppo del sistema, ed una volta effettuata non e' cambiabile dall'interno se il formalismo non ha adeguate capacita' di autorappresentazione. Viceversa, l'utilizzo di relazioni di ordine superiore, cioe' la possibilita' di esprimere proprieta' o relazioni di proprieta' o relazioni (e cosi' via, a vari livelli), mediante un linguaggio che abbia tali capacita' come RP, permette questo cambiamento. Gli esempi prima mostrati indicano che rappresentazioni diverse possono coesistere (in RP, espresse a livello base), e che il passaggio da una all'altra forma puo' esso stesso essere rappresentato e utilizzato (in RP, con le regole di metavalutazione). Questa duplice forma di metaconoscenza (proprieta' di proprieta' ... espresse a metalivello, e regole per il loro uso espresse a livello di metavalutazione) fornisce una potenza e una flessibilita' ben maggiori di quelle ottenibili da un qualunque formalismo 'piatto'.

Non secondario e' inoltre il ruolo cognitivo svolto dalla distinzione concettuale tra i livelli. Si

pensi ad esempio al fatto che i vari linguaggi di rappresentazione della conoscenza che incorporano modelli di reti semantiche si differenziano tra loro anche sul fatto di distinguere o meno tra la relazione di appartenenza di un elemento a una classe e la relazione di inclusione di una classe in un'altra (e piu' in generale sui diversi significati associati agli archi etichettati con 'is_a') [Br83]. Questi problemi 'semantici' si inquadrano diversamente una volta riconosciuta la necessita' di articolare ogni discorso non banale su piu' livelli. Interi campi teorici si dischiudono inoltre alle possibilita' di 'meccanizzazione', ad esempio il calcolo logico classico delle relazioni, che e' fatto di metarelazioni e di metateoremi. Un'esposizione sistematica di questa problematica sara' presentata in [CL90]. Esempi di applicazione di Reflective Prolog in altri settori, non riconducibili o difficilmente riconducibili a soluzioni Prolog, si trovano in [CL89a].

4. Confronto con altre proposte e considerazioni conclusive

Le citazioni riportate nell'introduzione mostrano che, sebbene nella pratica Prolog si sia rivelato in grado di far fronte alle necessita' delle applicazioni anche le piu' sofisticate, e presumibilmente anche per questo sia stato adottato e accettato quasi senza riserve da una parte della comunita' della programmazione logica, un'altra parte di essa ha analizzato criticamente il linguaggio allo scopo di migliorarlo, e ha posto apertamente il problema della sua evoluzione.

Reflective Prolog si inserisce in questa seconda tendenza, ed e' un primo risultato di un programma di ricerca, articolata sia sull'analisi di caratteristiche metalinguistiche adeguate [CL88] sia sugli strumenti tecnici per la loro implementazione [CCL89]. Il linguaggio, come qui descritto, e' pienamente definito ed implementato. In questo paragrafo lo collochiamo nel quadro di altre proposte in analoga direzione, quali risultano dalla letteratura piu' recente (un confronto con l'approccio 'classico' di Bowen e Kowalski [BK82] si trova in [CL89b]).

Uno dei piu' recenti approcci alla metaprogrammazione in programmazione logica e' il linguaggio HiLog [CKW89], che ha in sostanza lo scopo di incorporare le caratteristiche metalogiche 'impure' di Prolog, migliorando e semplificando la sintassi in modo da evitare l'uso di predicati metalogici come univ, args, eccetera. Riprendendo un esempio di [CKW89], la chiusura transitiva di una relazione binaria R, che si puo' definire in Prolog nel modo che segue:

```
chiusura(R,X,Y):-C=..[R,X,Y],call(C).
chiusura(R,X,Y):-C=..[R,X,Z],call(C),chiusura(R,Z,Y).
```

si definisce invece, in HiLog, come:

```
chiusura(R)(X,Y):-R(X,Y).
chiusura(R)(X,Y):-R(X,Z),chiusura(R)(Z,Y).
```

La sintassi e' molto piu' agile, e fa apparire HiLog come un linguaggio di ordine superiore, poiche' e' possibile usare liberamente variabili nelle posizioni proprie dei predicati senza l'utilizzo di alcun meccanismo di autoriferimento. La semantica di HiLog non e' pero' di ordine superiore, ma e' stata definita al prim'ordine utilizzando, anziche' il concetto di nome come in Reflective Prolog, un concetto di *intensione* associato a tutte le espressioni linguistiche. Ogni espressione ha un'intensione univoca, associabile pero' a diverse estensioni: cio' consente di interscambiare liberamente il ruolo degli oggetti (ad esempio lo stesso simbolo puo' fungere alternativamente, come in Prolog, da simbolo di predicato e di funzione). La semantica di HiLog, pur essendo espressa in logica del prim'ordine, e' pero' profondamente differente da quella classica dei linguaggi logici, e non esiste un concetto corrispondente a quello di modello minimo, che possa costituire un collegamento con la semantica procedurale.

D'altra parte HiLog non ha al momento una semantica procedurale precisa, e i programmi HiLog vengono eseguiti previa trasformazione in Prolog. In questo modo HiLog diventa una variante sintattica del Prolog (completo), e la trasformazione costringe a sacrificare diversi aspetti in linea di principio consentiti dalla semantica dichiarativa. In [CKW89] si argomenta che l'introduzione di una sintassi piu' appropriata e' gia' di per se' importante nella metaprogrammazione, perche' una diversa sintassi (motivata da una diverso modo di vedere) incoraggia un comportamento diverso da parte del programmatore, e puo' ispirare nuovi modi di pensare. Proprio in base a queste argomentazioni ci sembra pero' di poter osservare che una sintassi uniforme, che non permette di distinguere i diversi livelli di conoscenza e in particolare il livello oggetto dal livello meta, non incoraggia l'organizzazione della conoscenza e l'identificazione coerente delle diverse componenti di un metaprogramma.

Cio' che manca ad HiLog (come pure a Prolog) e' la possibilita', offerta in Reflective Prolog dalla riflessione, di definire delle vere e proprie regole di inferenza ausiliarie, che vengano automaticamente utilizzate quando necessario (lo stesso utilizzo di una variabile nella posizione di testa di una clausola, come ad esempio F(...):-... e' teoricamente possibile in HiLog, ma proceduralmente non gestibile). La procedura chiusura potrebbe essere utilizzata

ad esempio cosi' [CKW89]:

```
superiore_gerarchico(gianni,carlo).
superiore_gerarchico(carlo,mario).
obbedisce(X,Y):-chiusura(superiore_gerarchico)(X,Y).    /* in HiLog */
/* oppure */
obbedisce(X,Y):-chiusura(superiore_gerarchico,X,Y).    /* in Prolog */
?-obbedisce(gianni,Y).                                  /* Y=carlo e Y=mario */
```

Come si vede, e' necessario invocare esplicitamente chiusura quando si utilizza una procedura transitiva. In Reflective Prolog invece una possibile soluzione e' la seguente:

```
solve(#P($X,$Y)):-transitiva(#P),theory_fact(#P($X,$Z)),solve(#P($Z,$Y)).
transitiva(<superiore_gerarchico>).
obbedisce(X,Y):-superiore_gerarchico(X,Y).
```

La relazione superiore_gerarchico e' asserita come transitiva, e cio' fa in modo che una sottometa superiore_gerarchico(X,Y) non risolubile a livello base (si noti che non e' necessario gestire esplicitamente il caso in cui invece lo e') venga risolta mediante la clausola solve. Tale clausola dal punto di vista dichiarativo si puo' vedere come definizione della transitivita' di una relazione, e nel contempo proceduralmente realizza la chiusura transitiva e viene automaticamente applicata su qualsiasi relazione asserita come transitiva. Analogamente, si possono introdurre le piu' diverse proprieta' delle relazioni, come simmetria, riflessivita', equivalenza, difficili da realizzare sia in Prolog che in HiLog.

Un altro approccio importante nel campo delle estensioni dei linguaggi logici e' quello del λ-Prolog [Mi89], che introduce nel linguaggio delle clausole di Horn alcuni aspetti di second'ordine, sufficientemente deboli da permettere la definizione di una semantica procedurale (correlata pero' alla logica intuizionistica piuttosto che alla logica classica). Le estensioni apportate sono diverse, ma principalmente il λ-Prolog permette la presenza di formule atomiche quantificate universalmente e/o di implicazioni nelle condizioni delle clausole e nelle mete; permette inoltre di definire clausole (le cui variabili siano) quantificate esistenzialmente; le variabili possono infine spaziare (con qualche restrizione) su funzioni e predicati oltre che su individui. Lo scopo principale e' di introdurre meccanismi di strutturazione delle definizioni: localita' (nel senso ad esempio di rendere un predicato ausiliario locale alla definizione in cui viene utilizzato), modularizzazione e astrazione (e' possibile ad esempio definire tipi di dati astratti mediante insiemi di clausole quantificate esistenzialmente).

Le estensioni apportate in λ–Prolog sono molto diverse da quelle apportate in Reflective Prolog, sia dal punto di vista tecnico che delle finalita'. I due approcci non sono quindi in alcun modo in contrasto ma si possono anzi vedere come complementari, poiche' λ–Prolog offre una serie di meccanismi di strutturazione nell'ambito dei quali puo' trovare posto il modo diverso di rappresentazione della conoscenza sotteso dal Reflective Prolog. Il meccanismo di autoriferimento di RP, che prevede anche i nomi di predicato, consentirebbe probabilmente di rilassare alcuni degli aspetti di second'ordine del λ–Prolog.

In conclusione, consideriamo il Reflective Prolog come uno strumento di verifica di un insieme di istanze che si pongono nell'avanzamento della programmazione logica. Anche se molto lavoro resta ancora da fare, riteniamo che il Prolog evolvera' anche in questa direzione, e che, quando si sara' attestato e assestato a un livello superiore a quello attuale, la dicotomia menzionata nel sommario potra' ricomporsi.

Riferimenti

[ACS86] Aiello L., Cecchi C. e Sartini D., *Representation and Use of Metaknowledge*, Proceedings of the IEEE, vol. 74, n.10, October 1986, 1304–1321.

[BK82] Bowen K.A. e Kowalski R.A., *Amalgamating Language and Metalanguage in Logic Programming*, in: Clark K.L. and Tarnlund S–A. (eds.), *Logic Programming*, Academic Press, 1982.

[Br83] Brachman R.J., *What IS–A Is and Isn't: an Analysis of Taxonomic Links in Semantic Networks*, Computer, October 1983, 30–36.

[CCL89] Casaschi G., Costantini S. e Lanzarone G.A., *Realizzazione di un Interprete Riflessivo per clausole di Horn*, in: Mello P. (a cura di), atti del Quarto Convegno Nazionale sulla Programmazione Logica (gulp 89), Bologna, 7–9 Giugno 1989, 227–241.

[CKW89] Chen W., Kifer M., e Warren D.S., *HiLog: A First–Order Semantics for Higher–Order Logic Programming Constructs*, in: Proceedings of the 1989 North–American Conference on Logic Programming, Cleveland, Ohio, October 16–20, 1989.

[CL88] Costantini S. e Lanzarone G.A., *Un'architettura riflessiva per i linguaggi logici*, in: Nardi D. (a cura di), atti del Terzo Convegno Nazionale sulla Programmazione Logica (gulp 88), Roma, 11–13 Maggio 1988, 403–417.

[CL89a] Costantini S. e Lanzarone G.A., *Problem Solving in Metalogic Programming*, in: Proceedings of the IEEE Eighth Annual International Phoenix Conference on Computers and Communications, Scottsdale, Arizona, March 22–24, 1989, 543–548.

[CL89b] Costantini S. e Lanzarone G.A., *A Metalogic Programming Language*, in: Levi G. e Martelli M. (eds.), *Logic Programming*, Proceedings of the Sixth International Conference, MIT Press, 1989, 218–233.

[CL90] Costantini S. e Lanzarone G.A., *Metalogic Programming: Language, Semantics and Applications*, in preparazione.

[Co89] Costantini S., *Semantics of a Metalogic Programming Language*, Proceedings of the III Italian Conference on Theoretical Computer Science, Mantova, 2–4 Novembre 1989, World Scientific Publishing, 1989, 188–199.

[HL88] Hill P.M. e Lloyd J.W., *Analysis of Meta–programs*, in: Meta88, Proceedings of the Workshop on Meta–programming in Logic Programming, Bristol, 22–24 June, 1988, 27–42.

[Ll87] J.W. Lloyd, *Foundations of Logic Programming*, (Second, Extended Edition), Springer–Verlag, Berlin, 1987.

[Mi89] Miller D., *Lexical Scoping as Universal Quantification*, in: G. Levi e M. Martelli (eds.), *Logic Programming*, Proceedings of the Sixth International Conference, MIT Press, 1989, 268–283.

[MO84] Mycroft A. e O'Keefe R.A., *A Polymorphic Type System for Prolog*, Artificial Intelligence, vol.23, n.3, August 1984, 295–307.

[SS86] Sterling L. e Shapiro E., *The Art of PROLOG – Advanced Programming Techniques*, MIT Press, 1986.

[Wa82] Warren D.H.D., *Higher–order Extensions to Prolog: are They Needed ?*, in: Hayes J.E., Michie D. e Pao Y.H. (a cura di), Machine Intelligence n.10, Ellis Horwood, 1982, 441–453.

NOTE SULL'USO E LA DEFINIZIONE DI UN LINGUAGGIO PER LA PROGRAMMAZIONE LOGICA STRUTTURATA

Gianfranco Rossi
Dipartimento di Matematica e Informatica
Via Zanon 6 - UDINE

Sommario

Questo lavoro costituisce una continuazione ed ampliamento dei lavori descritti in [1], [2] e [3] e come tale si muove nella direzione di una piu' precisa definizione di un linguaggio logico esteso per la programmazione logica strutturata. In particolare, il lavoro affronta alcuni dei problemi lasciati aperti nei precedenti lavori, introduce una notazione sintattica adeguata per i costrutti di blocco, procedura e modulo forniti dal linguaggio, e mostra numerosi esempi d'uso del linguaggio stesso.

1. INTRODUZIONE

In precedenti lavori [1,2,3], abbiamo affrontato il problema di introdurre in un linguaggio logico strumenti adeguati per la strutturazione dei programmi. L'approccio da noi seguito, classificabile come approccio di tipo semantico, si basa essenzialmente sull'estensione delle clausole di Horn con i cosidetti ***goal implicazione***. Questa estensione consiste nel permettere che un generico goal G_i in una clausola A :- G_1,..., G_n possa essere non soltanto un atomo, ma anche un'implicazione $D \supset G$, dove G e' un goal e D e' un insieme di ***clausole locali*** a G (nel senso che possono essere utilizzate soltanto nella dimostrazione di G). L'idea di servirsi dei goal implicazione come strumento per avere definizioni locali di clausole e' stata utilizzata gia' in varie altre proposte, anche se con modalita' e con scopi diversi, tra cui ricordiamo quelle in [4], [5] e [6].

In [1] e [3], abbiamo mostrato che dando al goal implicazione ***diverse semantiche*** e' possibile ottenere diverse forme di visibilita' delle clausole definite come locali. In particolare, e' possibile definire sia ***ambienti aperti*** (in cui cioe' sono visibili anche clausole non locali), con regole di scope statiche, sia ***ambienti chiusi***, in modo analogo rispettivamente ai ***blocchi*** e ai ***moduli*** dei linguaggi di programmazione strutturati convenzionali.

In questo lavoro, ci poniamo l'obiettivo di definire piu' precisamente un ***linguaggio di programmazione*** logica (in realta' un Prolog esteso) basato sulle idee esposte nei lavori precedenti sopra citati. A tal scopo, affronteremo alcuni dei problemi lasciati aperti nei precedenti lavori ed introdurremo una *notazione sintattica* adeguata per il linguaggio logico esteso, mostrando numerosi *esempi* d'uso del linguaggio stesso. In particolare, nel cap.2 vengono precisate meglio le *regole di scope* per le variabili e le clausole per il caso dei blocchi; nel cap.3 viene mostrato come i blocchi possano usarsi facilmente per realizzare l'astrazione di *procedura*; nel cap.4 viene precisata la definizione di modulo, con la relativa specifica di *interfaccia*, e mostrato l'uso di *moduli parametrici*. Infine viene descritta una possibile *implementazione* del linguaggio con i blocchi tramite traduzione in Prolog.

Lavoro parzialmente finanziato da C.N.R. - Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo".

2. BLOCCHI

2.1 Definizione di blocco

In [1] abbiamo mostrato che e' possibile dare ad un goal implicazione una semantica che permette di avere per le clausole locali le solite regole di scope dei blocchi dei linguaggi convenzionali Algol-like. Un goal di questo tipo, che indicheremo anche come ***goal blocco***, puo' essere definito utilizzando la seguente notazione sintattica:

{c1.c2. ... .cm} => (g1,...,gn).

c1,...,cm rappresentano le clausole locali al blocco e g1,...,gn i goal che utilizzano queste clausole (e cioe' il "corpo" del blocco). Le regole di scope per i blocchi prevedono che le clausole dichiarate in un blocco B siano visibili in B e in tutti i blocchi contenuti in B, ma non nei blocchi esterni a B. L'ambiente definito da un blocco e' percio' ***aperto*** nel senso che in esso si possono utilizzare sia le clausole locali che quelle definite negli eventuali blocchi che lo contengono. L'insieme delle clausole che puo' essere utilizzato nella prova di un goal dipende percio' soltanto dalla struttura lessicale del programma (regole di scope statiche).

Come esempio di riferimento per la presentazione del linguaggio con i blocchi, considereremo la definizione di un semplice programma per la visita di un grafo diretto (eventualmente con cicli) rappresentato al solito come una serie di asserzioni del tipo *a(n1,n2)* dove *n1* ed *n2* sono due nodi distinti del grafo (per la definizione in Prolog si veda ad esempio [8, p.159]). Altri esempi piu' complessi si possono trovare in Appendice.

Esempio 1.

```
go(X,X,T).
go(X,Y,T) :-  a(X,Z),
                  {legal(X,[]).
                   legal(X,[H|T]) :- X\==H,legal(X,T)} =>
              legal(Z,T),
              go(Z,Y,[Z|T]).
```

In questo esempio, si e' supposto che *legal* venga utilizzato soltanto da *go* e si e' percio' definito *legal* in un blocco interno a *go* stesso (e quindi non visibile nell'ambiente piu' esterno). Si suppone inoltre che il predicato *a* che rappresenta il grafo su cui operare sia definito nell'ambiente piu' esterno.

Volendo rendere "trasparente" all'utente la presenza del terzo parametro nella definizione di *go*, potremmo definire un nuovo predicato, *path(X,Y)*, contenente la definizione del predicato *go* come blocco interno, nel modo seguente:

Esempio 2.

```
path(X,Y) :-
        {go(X,X,T).
         go(X,Y,T) :-  a(X,Z),
                           {legal(X,[]).
                            legal(X,[H|T]) :- X\==H,legal(X,T)} =>
                       legal(Z,T),
                       go(Z,Y,[Z|T])} =>
    go(X,Y,[]).
```

In questo esempio si nota che e' possibile avere ***blocchi annidati***, ad un livello di annidamento qualsiasi. Da notare inoltre che sebbene il predicato *a* sia definito nell'ambiente piu' esterno (quello contente la definizione di *path*) esso risultera' comunque visibile dal blocco piu' interno grazie alle regole di visibilita' per i blocchi previste dal linguaggio.

2.2 Definizione di variabili locali

Un linguaggio logico con i blocchi richiede che lo scope delle variabili sia specificato in modo preciso. Infatti, dal momento che la definizione di una clausola puo' comparire nel corpo di un'altra clausola, e' necessario distinguere tra le ***variabili locali*** ad essa e quelle ***globali***, cioe' provenienti dall'ambiente esterno. L'*uso esplicito dei quantificatori* permette di dare facilmente delle semplici regole di scope statico alle variabili di un programma logico. Precisamente, lo scope di $\forall x$ (risp. $\exists x$) in $\forall xF$ (risp. $\exists xF$) e', come al solito, la formula F.

D'altra parte, la necessita' di specificare i quantificatori per tutte le variabili risulta piuttosto pesante. Per semplificare la scrittura dei programmi, si potrebbe assumere che tutte le variabili siano *universalmente quantificate in modo implicito* per ciascuna clausola, come nel caso del Prolog (e come assunto, per semplicita', negli Esempi 1 e 2). In questo modo, pero', si perde completamente la possibilita' di utilizzare variabili globali cosi' come la possibilita' di definire variabili locali a uno o piu' goal.

La soluzione da noi adottata e' percio' intermedia tra questi due estremi. Infatti, nel nostro linguaggio si assume che tutte le variabili nella testa di una clausola siano quantificate ***universalmente*** su tutta la clausola in modo ***implicito***, mentre tutte le variabili che compaiono soltanto nel corpo di una clausola e che sono ***locali*** alla clausola stessa devono essere quantificate ***esistenzialmente*** in modo ***esplicito*** rispetto ad un goal o ad una congiunzione di goal (in particolare, rispetto ad un goal blocco). Altre variabili (cioe' variabili che non appaiono nella testa di una clausola o che non sono esplicitamente quantificate) sono considerate come ***globali***.

Per aumentare la leggibilita' dei programmi e semplificarne altressi' la loro analisi ed interpretazione, viene utilizzata la seguente notazione per definire variabili locali:

localvars [X1,...,Xn]: g1,g2,...,gm   con $n,m \geq 1$ (1)

dove X1,...,Xn rappresentano variabili quantificate esistenzialmente su g1,...,gm (se c'e' una sola variabile locale, non e' richiesto di utilizzare la notazione a lista). L'unica limitazione che pone questa soluzione e' che non e' piu' possibile avere variabili globali nella testa di una clausola. Ad esempio, la clausola ∀X(p(X,Y):-q), con Y variabile globale, deve essere riscritta nel nostro linguaggio come p(X,Z):-Z=Y,q.

Utilizzando queste convenzioni, l'Esempio 1 puo' essere riscritto nel modo seguente:

Esempio 3.

```
go(X,X,T).
go(X,Y,T) :- localvars Z:
        a(X,Z),
             {legal([]).
              legal([H|T]) :- Z\==H,legal(T)} =>
        legal(T),
        go(Z,Y,[Z|T]).
```

La variabile **Z** nella seconda clausola del predicato *legal* e' chiaramente una ***variabile globale***. Va comunque osservato che l'impiego di variabili globali, sebbene utile in varie applicazioni pratiche, puo' risultare notevolmente dannoso per quanto riguarda la comprensibilita' dei programmi, analogamente a quanto accade nei linguaggi di

---

(1) localvars/1 e' trattato come un operatore prefisso che viene applicato a un termine con funtore principale l'operatore binario infisso :/2 il cui secondo argomento e' una congiunzione di goal e il primo argomento e' la lista delle variabili locali a questi goal.

programmazione convenzionali. L'uso di variabili globali va percio' limitato e comunque fatto con una certa attenzione.

2.3 Definizione di predicati

Come nei linguaggi di programmazione convenzionali, le regole di scope per le clausole sono complicate dal fatto che un predicato in un blocco B puo' avere lo stesso nome di un predicato definito in un eventuale blocco contenente B. La convenzione nei linguaggi Algol-like e' che le dichiarazioni locali "ricoprano" quelle globali. In un programma logico, invece, la situazione puo' essere piu' complessa dato che la definizione di un predicato puo' essere data per mezzo di piu' clausole che potrebbero trovarsi in blocchi diversi (si noti che qui si assume, come del resto avviene nella maggior parte delle proposte, che i nomi dei predicati siano sempre globali, anche se le clausole sono locali).

La semantica del linguaggio con i blocchi data in [1] prevede che tutte le clausole visibili che definiscono un predicato p possano essere usate per risolvere un dato goal; precisamente, una delle definizioni che soddisfano il goal viene scelta in modo ***non-deterministico***. In questo modo, la definizione di un predicato data in un blocco viene ***estesa*** con le definizioni dello stesso predicato date nei blocchi piu' esterni. (2) Nell'implementazione concreta del linguaggio, il non-determinismo presente nella definizione data in [1] e' stato risolto assumendo di procedere sempre dai blocchi piu' interni verso quelli piu' esterni, utilizzando il solito meccanismo di backtracking per passare da una alternativa alla successiva in caso di fallimento.

L'*estensione dei predicati* puo' essere utile ad esempio per trattare le eccezioni o i casi particolari di una regola generale. La regola generale e' definita nell'ambiente piu' esterno; i casi particolari e le eccezioni sono definite in un blocco piu' interno e sono aggiunti alla regola generale soltanto quando si entra nel blocco. Ad esempio

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
 ...
p :- {member('$',L) :-!.
      member('#',L) :- !. ...} => member(X,L), ... .
```

e cioe', i caratteri speciali sono considerati come appartenenti sempre alla lista data, anche se essi non compaiono esplicitamente in essa. Da notare che il cut usato nel blocco interno si suppone abbia effetto a livello globale e cioe' anche sulle altre definizioni del predicato *member* contenute nel blocco piu' esterno.

La scelta tra estensione o ricoprimento dei predicati e' comunque ancora da indagare adeguatamente. Probabilmente sarebbe utile avere entrambe le possibilita'. In Contextual L.P. [6,11], ad esempio, si ha ricoprimento come default ma e' possibile richiedere anche estensione per un predicato tramite apposita dichiarazione. Il problema e' comunque al solito quello di trovare una semantica adeguata per entrambe le possibilita'. Un'ipotesi che si sta indagando per il nostro linguaggio e' quella di usare i quantificatori anche sui nomi dei predicati in modo analogo a quanto proposto da Miller in [16] per far si che anche i nomi dei predicati possano avere uno *scope* locale e non soltanto globale.

Per ora, come "trucco implementativo", se si vuole evitare che la definizione di un predicato p/n in un blocco B venga estesa si puo' aggiungere alla definizione di p in B

---

(2) Viceversa, se si applicasse una regola di *ricopertura dei predicati* ("predicate overriding"), un predicato *p* definito in un certo blocco cancellerebbe tutte le definizioni di *p* eventualmente contenute in blocchi piu' esterni (cfr. ad esempio [11]).

un'ultima clausola del tipo

```
p(X1,...,Xn) :- !,fail
```

che permette di "tagliare" tutte le eventuali alternative successive. Ovviamente questa aggiunta potrebbe essere mascherata all'utente con un'opportuna dichiarazione sintattica e l'implementazione potrebbe evitare, in taluni casi, l'effettiva inserzione di questa clausola aggiuntiva pur garantendo il funzionamento previsto dalla sua presenza.

3. PROCEDURE

Nei linguaggi di programmazione convenzionali uno strumento importante per la strutturazione dei programmi e' rappresentato dai *sottoprogrammi* (procedure e funzioni). Come i blocchi, anche i sottoprogrammi permettono di definire un ***ambiente locale*** raggruppando insieme piu' dichiarazioni di variabili e i relativi statement. A differenza dei blocchi, pero', i sottoprogrammi hanno un ***nome*** e dei ***parametri***. Inoltre, un sottoprogramma e' eseguito soltanto quando viene richiamato esplicitamente tramite il suo nome.

In Prolog, come noto, una sequenza di clausole aventi tutte lo stesso predicato di testa con la stessa arita' e' considerata come la definizione di una procedura. L'unificazione e' usata come unico meccanismo per il passaggio dei parametri. Il Prolog, comunque, non fornisce nessun supporto per raggruppare insieme tutte le clausole che definiscono una procedura. Non e' neanche possibile definire altre procedure e/o variabili come locali ad una procedura.

Il meccanismo dei blocchi presentato nel capitolo precedente fornisce una semplice soluzione al problema della definizione di procedure in un linguaggio logico. Un blocco puo' venir usato per definire un ambiente locale in cui le clausole che definiscono una procedura con le relative dichiarazioni locali possono venir ragruppate insieme e nascoste all'ambiente esterno. Piu' precisamente, una procedura p/n, composta da m clausole $p(a_1,...,a_n)$:-β_1,..., $p(b_1,...,b_n)$:-β_m, con procedure locali $\alpha_1,...,\alpha_k$ ($n,k \geq 0$, $m \geq 1$), puo' essere definita nel modo seguente:

$$
\begin{array}{l}
p(X_1,...,X_n) :- \\
\quad \{a_1. \\
\quad\ ... \\
\quad a_k. \\
\quad p(a_1,...,a_n) :- b_1. \\
\quad\ ... \\
\quad p(b_1,...,b_n) :- b_m. \\
\quad p(Y_1,...,Y_n) :- !,fail\ \} \Rightarrow p(X_1,...,X_n).
\end{array}
$$

La presenza dell'ultima clausola permette di considerare un predicato definito come procedura come un predicato non estendibile. Infatti, il cut permette di tagliare tutte le eventuali alternative del predicato *p/n* non ancora tentate presenti nell'ambiente contente la definizione della procedura o in un blocco piu' esterno. In perticolare, il cut evita che si tenti la definizione piu' esterna del predicato *p/n* dopo aver tentato tutte quelle interne alla procedura (come previsto dalla regola di estensione dei predicati), finendo cosi in un ciclo senza fine. E' comunque chiaro che in questo caso sarebbe sicuramente opportuno avere la modalita' di *ricopertura dei predicati* per il predicato *p/n*.

Per semplificare l'uso delle procedure e' stata introdotta nel linguaggio una ***notazione sintattica*** che rende la definizione di procedura piu' concisa. Ad es., la procedura p/n mostrata sopra puo' venir definita nel modo seguente:

```
p(X1,...,Xn) isproc
      {a1.
       ...
       ak.
       p(a1,...,an):-b1.
       ...
       p(b1,...,bn):-bm}.
```

Come esempio di uso delle procedure, consideriamo di nuovo il predicato *go/3*, ma esteso a grafi non direzionali. La sua definizione in Prolog puo' essere (cfr. ad es. [8, p.139]):

```
go(X,X,T).
go(X,Y,T)  :-  a(X,Z),legal(Z,T),go(Z,Y,[Z|T]).
go(X,Y,T)  :-  a(Z,X),legal(Z,T),go(Z,Y,[Z|T]).
```

In questo caso, il predicato *legal* e' richiamato in due punti diversi, e quindi mal si presta ad essere semplicemente definito in un blocco interno. Viceversa, *legal* puo' essere convenientemente definito come una procedura, mantenendo inalterate le modalita' della sua chiamata all'interno di *go*. Anche il predicato *go* puo' essere a sua volta definito come procedura, con *legal* visto come sua procedura locale (le procedure possono essere annidate a qualsiasi livello, come i blocchi), ottenedo cosi' la seguente definizione:

Esempio 4.

```
go(X,Y,T) isproc
      {legal(X,T) isproc      {legal(X,[]).
                                legal(X,[H|T]) :- X\==H,legal(X,T)}.
       go(X,X,T).
       go(X,Y,T) :- localvars Z:
             a(X,Z),legal(Z,T),go(Z,Y,[Z|T]).
       go(X,Y,T) :- localvars Z:
             a(Z,X),legal(Z,T),go(Z,Y,[Z|T])}.
```

Si noti in questo esempio che essendo ora la definizione di *legal* non piu' locale alla singola clausola del predicato *go* bensi' locale all'intero blocco che contiene il corpo di *go* non e' piu' possibile usare la variabile Z come globale in essa. Si noti inoltre che definire *legal* come procedura locale e' un modo per renderlo non estendibile rispetto ad altre eventuali definizioni del predicato *legal* presenti nell'ambiente piu' esterno.

Per quanto riguarda l'implementazione del costrutto di procedura, e' chiaro che esso non deve essere necessariamente implementato come la definizione e richiamo di un blocco nel modo visto sopra. Questo schema definisce la semantica del costrutto, ma la sua implementazione concreta puo' essere diversa e possibilmente piu' efficiente. In particolare, essendo tutto statico, e' facile per un'implementazione efficiente evitare l'unificazione in piu' richiesta per l'attivazione del blocco contenente il "corpo" della procedura.

4. MODULI

4.1 Definizione di modulo

Oltre alla possibilita' di definire blocchi, il nostro linguaggio da' anche la possibilita' di definire ***moduli***. Come nei linguaggi convenzionali, un modulo consiste in un insieme di definizioni locali al modulo stesso, a cui e' possibile riferirsi tramite un ***nome***. A differenza dei blocchi, i moduli costituiscono un ***ambiente chiuso*** di definizioni di predicati, da cui cioe' non si ha alcuna visibilita' dell'ambiente esterno.

Anche l'introduzione dei moduli, come gia' quella dei blocchi, si basa sull'estensione delle clausole di Horn con goal implicazione a cui viene data un'opportuna semantica. Oltre a cio', viene introdotto nel linguaggio un operatore, chiamato *ismod*, per associare un nome ad un modulo: ***M ismod D*** definisce un modulo di nome M costituito dall'insieme di clausole D. Un goal implicazione che permette di dimostrare un goal G nel modulo identificato da M e' rappresentato in questo caso come ***demo(M,G)***. In prima approssimazione, si puo' dire che un goal *demo(M,G)* e' dimostrabile in un programma P se il goal G e' dimostrabile nel programma associato ad M. Percio' la semantica di un modulo e' completamente indipendente dall'ambiente esterno.

Un ***programma*** e' definito come una coppia <E,D> consistente in un insieme D di clausole e in un insieme E di definizioni di moduli. I nomi dei moduli sono globali e quindi i moduli componenti un programma sono tutti visibili tra loro. In questa versione del linguaggio, si e' esclusa la possibilita' di definire moduli all'interno di altri moduli (moduli annidati), mentre e' possibile definire blocchi (in particolare procedure) all'interno di moduli..

Esempio 5. (cfr.[11])

```
lists  ismod
      {member(X,[X|_]).
       member(X,[_|L]) :- member(X,L).
       append([],L,L).
       append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
       ...
       sort(List,Sorted) :- localvars List1:
                 swap(List,List1), sort(List1,Sorted).
       sort(Sorted,Sorted).

       swap([X,Y|Rest],[Y,X|Rest]) :- demo(intord,gt(X,Y)).
       swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
       ...}.

intord ismod
      {gt(X,Y) :- ...
       lss(X,Y) :- ...
       equal(X,Y) :- ...}.

?- demo(lists,sort([3,1,2],Sorted)).
```

Tutti i predicati definiti in un modulo sono visibili al suo esterno. L'"***import***" di uno di questi predicati in un altro modulo puo' avvenire pero' soltanto tramite *demo* e quindi specificando in modo esplicito il nome del modulo da cui si importa. In questo modo si ottiene una ***configurazione statica*** di moduli, simile a quella ottenibile con i linguaggi di programmazione convenzionali che prevedono moduli, come ad es. Ada. Questo aspetto di staticita' facilita ovviamente l'implementazione del linguaggio: e' infatti sempre possibile legare un predicato alla sua definizione a "compile-time" anche se il predicato e' definito in un altro modulo. Questo non e' possibile, al contrario, nel Contextual L.P. di Monteiro e Porto [6] dato che il legame tra un predicato usato in un modulo m1 e la sua definizione in un modulo m2 puo' essere stabilito soltanto dinamicamente nel momento in cui il contesto di m1 e' esteso con m2. Per ovviare a questi problemi ed ottenere cosi' implementazioni piu' efficienti, in [7] e [11] viene proposta un'estensione del Contextual L.P. che permette di definire configurazioni statiche di moduli come nel nostro linguaggio.

Un modo per avere una forma di configurazione dinamica dei moduli anche nel nostro linguaggio e' quello di permettere la definizione di moduli parametrici, argomento di cui

tratteremo nel cap. 4.3.

4.2 Interfaccia dei moduli

Per limitare la visibilita' dei predicati definiti in un modulo, si puo' far ricorso all'utilizzo di blocchi interni al modulo, come gia' suggerito in [2]. Cosi', se p1,...,pn,q1,...,qm sono predicati definiti in un modulo *alfa* e si vuole che soltanto i predicati p1,...,pn siano esportati da *alfa*, allora si puo' definire *alfa* nel modo seguente:

```
alfa ismod
    {p1:-  {q1:-g1.
            ...
            qm:-gm} => b1.
     ...
     pn:-  {q1:-g1.
            ...
            qm:-gm} => bn}.
```

dove βi e γi rappresentano generiche congiunzioni di goal. In questo modo, i predicati q1,...,qm non sono visibili all'esterno del modulo *alfa*.

Un po' di "zucchero sintattico" puo' essere usato per rendere la definizione di modulo piu' "elengate". Per esempio, la definizione del modulo *alfa* data sopra puo' essere riscritta nel modo seguente:

```
alfa ismod
    {p1:-b1.
     ...
     pn:-bn}
body
    {q1:-g1.
     ...
     qm:-gm}.
```

che e' equivalente alla definizione precedente, ma piu' leggibile.

Nel programma dell'Esempio 5 puo' essere opportuno definire il predicato *swap* come non visibile all'esterno del modulo *lists*. Pertanto, *swap* puo' essere definito nella parte body del modulo *lists* nel modo indicato schematicamente qui di seguito:

```
lists ismod
    {member(X,[X|_]).
      ...
      sort(Sorted,Sorted)}.
body
    {swap([X,Y|Rest],[Y,X|Rest]) :- demo(intord,gt(X,Y)).
      ...}.
```

E' importante rendersi conto della sostanziale differenza che esiste tra i due operatori ismod e demo e gli altri operatori come localvars, isproc, body. I primi due possono esser visti come operatori modali facenti parte del linguaggio logico esteso e quindi semanticamente significativi; gli altri operatori, invece, sono soltanto zucchero sintattico introdotto al solo scopo di rendere i programmi piu' facilmente leggibili.

Infine, notiamo che, essendo possibile avere blocchi all'interno di moduli, puo' essere conveniente, dal punto di vista della leggibilita' dei programmi, utilizzare il costrutto di

procedura per definire i predicati esportati da un modulo. Nel modulo *lists*, ad esempio, le due clausole della *member* potrebbero venir rimpiazzate dalla definizione di procedura member(X,L) isproc {...} (ved. anche l'Esempio 5 in Appendice).

4.3 Moduli parametrici

Nella nostra proposta (cfr. [3]), l'insieme D delle clausole definite all'interno di un modulo *M ismod D* non e' necessariamente un insieme chiuso di clausole; D puo' cioe' contenere delle ***variabile libere*** (diversamente ad esempio dalle ***teorie*** della proposta di Bowen e Kowalski [14]). Queste variabili sono ammesse al solo scopo di permettere la definizione di ***moduli parametrici.*** Si richiede pertanto che le variabili libere che eventualmente occorrono in D siano presenti anche come argomenti nel nome del modulo M [(3)] (dunque M e' un formula atomica, non un nome costante). Queste variabili sono da considerarsi universalmente quantificate in modo implicito davanti alla definizione del modulo stesso.

Come parametro di un modulo e' possibile specificare un termine qualsiasi, oppure un nome di predicato o un nome di modulo (in tutti i casi deve trattarsi di una formula atomica ***ground*** [(3)]). Il secondo e terzo caso richiedono di passare ad una logica ***higher-order***, anche se in realta' come mostrato in [3], e' sufficiente una forma molto limitata di higher-order che tra l'altro non intacca la decidibilita' dell'unificazione. I nomi di predicati o di moduli passati come parametri ad un modulo parametrico possono essere usati soltanto in goal del tipo *demo(M,G)* (in *M* o in *G*).[(3)] In questo modo un modulo puo' essere parametrico soltanto rispetto a predicati definiti nell'ambiente esterno (ed importati esplicitamente tramite *demo*) e non rispetto a predicati definiti al suo interno.

Come primo esempio possiamo considerare di nuovo il modulo dell'Esempio 5 e ridefinirlo in modo da renderlo parametrico rispetto al modulo che definisce la relazione di ordinamento utilizzata dal predicato *swap*.

Esempio 6.

```
lists(Ord) ismod
    { ...
     swap([X,Y|Rest],[Y,X|Rest]) :- demo(Ord,gt(X,Y)).
     swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
     ...}.

?-  demo(lists(intord),sort([3,1,2],Sorted)).
    Sorted = [1,2,3]
?-  demo(lists(charord),[b,c,a],Sorted).
    Sorted = [a,b,c].
```

dove *charord* e' un modulo che definisce i predicati *gt/2*, *equal/2*, ... sui caratteri invece che sugli interi come invece avviene per *intords*.

In generale, la possibilita' di specificare un nome di modulo come parametro, permette fra l'altro di modificare facilmente e in modo dinamico l'implementazione di un predicato *p* senza modificarne l'uso. Per ottenere cio', e' necessario rendere il modulo *m* in cui *p* e' usato parametrico rispetto al nome del modulo in cui *p* e' definito:

---

(3) Queste proprieta' vengono garantite imponendo opportune ***restrizioni sintattiche*** sulla posizione delle variabili all'interno di un modulo.

```
m(M) ismod
     {q:-demo(M,p),...
      ...}.
```

Supponendo che m1 ed m2 contengano implementazioni diverse di *p*, possiamo richiedere alternativamnte l'uso dell'una o dell'altra implementazione eseguendo uno dei due goal:

```
?- demo(m(m1),q)
?- demo(m(m2),q).
```

Il modulo in cui cercare la definizione di *p* e' ora specificato a run-time e puo' essere cambiato senza modificare il modulo *m* (analoga considerazione vale per il predicato *gt/2* nell'Esempio 6). Si ha dunque la possibilita' di definire una ***configurazione dinaminica*** di moduli. Questa possibilita' e' di norma garantita dalle proposte che considerano i moduli come ambienti aperti (ad es. [5] e[6]); rispetto a queste, l'uso di moduli chiusi con parametri, come quelli della nostra proposta, permette di dare ad un modulo visibilita' dell'ambiente esterno in modo piu' "controllato" sebbene dinamico (l'"import" di un predicato avviene infatti sempre tramite *demo*). D'altra parte, rispetto al Contextual L.P. (nella sua versione estesa con la nozione di "lazy binding" presentata in [7,11]) e anche rispetto alla proposta di Miller [5], col nostro linguaggio non e' possibile dare ***definizioni ricorsive*** di moduli, parametrici l'uno rispetto all'altro.

Nella nostra proposta e' possibile avere anche il ***nome di un predicato*** come parametro di un modulo. Con riferimento all'Esempio 6, questa ulteriore possibilita' permette di rendere l'operazione di *sort* parametrica anche rispetto al tipo di ordinamento che si vuol dare alla lista. La nuova definizione di *lists* diventa ora la seguente:

Esempio 7.

```
lists(Ord,Rel/2) ismod
     { ...
      swap([X,Y|Rest],[Y,X|Rest]) :- demo(Ord,Rel(X,Y)).
      ...}.

?-  demo(lists(intord,lss),sort([3,1,2],Sorted)).
    Sorted = [3,2,1]
```

Si noti che, insieme ad una variabile predicativa usata come parametro di un modulo, viene specificata anche l'arita' del predicato a cui la variabile si riferisce (l'arita' puo' essere specificata anche come variabile che deve comunque essere istanziata ad una costante al momento della chiamata del modulo). La presenza di questa informazione aggiuntiva e' dovuta al fatto che un predicato in Prolog e' caratterizzato non soltanto dal suo nome ma anche dalla sua arita', e ha inoltre lo scopo di facilitare l'implementazione dei moduli parametrici.

5. NOTE DI IMPLEMENTAZIONE

L'implementazione del linguaggio fin qui descritto si puo' ottenere partendo dalla semantica operazionale data in [1] e [3] per giungere, attraverso raffinamenti successivi, ad interpreti via via piu' concreti. Una simile metodologia e' descritta ad esempio in [11] per il caso di un linguaggio di programmazione logica convenzionale ed e' stata applicata alo sviluppo di EnvProlog [12]. Alcuni suggerimenti riguardo l'implementazione del linguaggio con i blocchi secondo questo approccio sono contenuti nell'ultimo cap. di [1].

La natura statica del nostro linguaggio favorisce la realizzazione di implementazioni efficienti basate sulla ***compilazione***, in analogia con quanto avviene per i linguaggi di programmazione convenzionali. In realta', la natura statica del nostro linguaggio permette di implementare il linguaggio stesso semplicemente tramite traduzione diretta (***preprocessing***) in

Prolog standard.(4) Qui ci limiteremo ad illustrare, in modo intuitivo, l'applicazione di questa tecnica al caso del linguaggio con i blocchi. Una soluzione simile e' stata proposta per un linguaggio simile al nostro da Moscowitz e Shapiro in [9]. Contrariamente a quanto affermato da questi autori, sembra possibile tradurre il nostro linguaggio con i blocchi direttamente in Prolog senza dover introdurre in esso nuove restrizioni.

La tecnica da noi utilizzata e' la seguente. I blocchi sono individuati in modo univoco ognuno da una costante b_i ($i \geq 0$), con b_0 che identifica il blocco piu' esterno. Al predicato di testa di ciascuna clausola si aggiunge (ad esempio come primo parametro) un termine che rappresenta l'annidamento (statico) del blocco, cui la clausola appartiene, all'interno del programma. Il termine $b_0(X)$ (dove X e' una variabile che non compare gia' nella clausola) indica che ci troviamo nel blocco piu' esterno; i termini $b_0(b_1(X))$, ..., $b_0(b_k(X))$ indicano rispettivamente i k blocchi di primo livello, e cosi' via. In generale, il sottotermine $...b_i(b_j(...))$ indica che il blocco b_j e' contenuto nel blocco b_i (per semplicita', b_0 viene omesso per cui il blocco piu' esterno sara' indicato semplicemente dal termine variabile X). Analogamente, a ciascun goal che appare nel corpo di una clausola viene aggiunto un termine di questo tipo (ma con X rimpiaxxata da una costante c), per indicare a quale blocco il goal stesso appartiene. Consideriamo ad esempio il seguente programma:

```
q.
s :- {p :-  {t :- r,q.
             q} => t, q.
      r} => p.

?- s.
yes
```

La traduzione in Prolog di questo programma e' la seguente:

```
s(X) :- p(b1(c)).
p(b1(X)) :- t(b1(b2(c))),q(b1(c)).
r(b1(X)).
t(b1(b2(X))) :- r(b1(b2(c))),q(b1(b2(c))).
q(b1(b2(X))).
q(X).

?- s(c).
yes
```

Con questa tecnica, il rispetto delle regole di scope previste per i blocchi e' imposto dalla possibilita' o meno di unificazione tra i diversi termini aggiunti ai goal e predicati di testa delle clausole. Ad esempio, la chiamata di *q* all'interno del blocco b2 unifica sia con la definizione di *q* data nello stesso blocco, sia con la definizione di *q* data in b0, cioe' in un blocco piu' esterno; l'altra chiamata di *q* invece unifica soltanto con la definizione di *q* in b0, essendo quella in b2 contenuta in un blocco piu' interno e quindi non visibile. Da notare che nel codice Prolog le due definizioni di *q* sono date in modo tale che prima si provi quella del blocco piu' interno (b2) e poi l'altra (tramite backtracking). La presenza di eventuali cut all'interno di un blocco non dovrebbe presentare problemi con la tecnica d'implementazione dei blocchi sopra descritta. I cut sono considerati sempre come ***cut globali*** , relativi cioe' a tutte e sole le definizioni dei predicati visibili dal punto in cui compare il cut; cio' e' garantito dall'ordinamento delle clausole nel codice Prolog generato e dall'uso dell'unificazione per discriminare tra predicati

---

(4) Nel caso di implementazione basata su WAM, le stesse considerazioni ci permettono di dire che il trattamento dei blocchi puo' avvenire tutto in fase di compilazione, senza richiedere alcuna modifica alla WAM standard.

appartenenti a blocchi distinti (che evita di tagliare alternative appartenenti a blocchi non visibili).

Il trattamento delle ***variabili*** locali e globali nella traduzione dal nostro linguaggio al Prolog non e' ancora stato analizzato precisamente, ma non dovrebbe presentare grosse difficolta', come del resto mostrato anche da Moscowitz e Shapiro in [9] per un linguaggio simile al nostro linguaggio con i blocchi.

Per quanto riguarda i ***moduli***, anche in questo caso risulta facilmente fattibile la loro implementazione tramite traduzione in Prolog, in modo analogo a quanto visto per i blocchi. Questo tipo di implementazione per i moduli, sebbene facile da realizzare, non risulta pero' molto soddisfacente dal punto di vista dell'efficienza del codice generato a causa dell'aumento del numero degli argomenti di ciascun predicato e del conseguente appesantimento delle operazioni di ricerca della clausola da selezionare. In questo caso, sarebbe senz'altro piu' opportuno tradurre il linguaggio con blocchi e moduli non in Prolog standard bensi' in un Prolog esteso che offra meccanismi built-in per definire clausole locali, come ad esempio ***EnvProlog*** [12] (con le sue possibilita' per la definizione e manipolazione di programmi come dati [13]) o il Contextual L.P. esteso [7,11].

6. CONCLUSIONI E LAVORI FUTURI

In questo lavoro abbiamo presentato un linguaggio per la ***programmazione logica strutturata***, dotato di costrutti per definire blocchi, procedure e moduli parametrici, basato sulle idee presentate in [1,2,3,10]. Avendo ormai completata la definizione di una parte significativa del linguaggio, stiamo ora affrontando i problemi relativi alla sua ***implementazione***, seguendo l'approccio delineato nell'ultimo capitolo di questo lavoro. In particolare, nel prossimo futuro, sara' disponibile l'implementazione per il linguaggio esteso *completo*, basata sulla traduzione del linguaggio stesso in ***C_Prolog*** (con possibilita' anche di "decompilazione" da C_Prolog a linguaggio esteso). In parallelo, si sta definendo un'analoga implementazione rivolta pero' al ***Contextual L.P.*** messo a punto presso il DEIS di Bologna [7,11], implementazione che, rispetto a quella in C_Prolog, dovrebbe risultare notevolmente piu' efficiente potendo sfruttare il meccanismo delle *unita'* built-in nel linguaggio per definire clausole locali.

Riguardo alla definizione del linguaggio esteso, verra' ancora indagata la possibilita' (e la convenienza) di usare l'approccio proposto in [15] per descrivere la semantica del nostro linguaggio (completo). Nel far questo, si indaghera' anche la possibilita' di dare una definizione e una semantica piu' soddisfacente per i moduli parametrici, nonche' la possibilita' di inserire nel nostro linguaggio anche operazioni di ***composizione dinamica*** di moduli.

RIFERIMENTI BIBLIOGRAFICI

[1] Giordano L., Martelli A., Rossi G.F.: "Local definitions with static scope rules in Logic Languages", *Int. Conf. on Fifth Generation Computer Systems*, Tokyo, Nov. 1988., 389-396.

[2] Giordano L., Martelli A., Rossi G.F.: "Una proposta per l'introduzione di blocchi e moduli nei linguaggi logici", *Quarto Convegno Nazionale sulla Programmazione Logica*, Bologna, 1989.

[3] Giordano L., Martelli A., Rossi G.F.: "Extending Horn Clause Logic with Module Costructs", Rapporto di Ricerca 04-90-RR, Dip. di Matem. e Informatica, Univ. di Udine, 1990.

[4] Gabbay D.M., Reyle N.: "N_Prolog: An Extension of Prolog with Hypothetical Implications.I.", *Journal of Logic Programming*, no.4 1984, 319-355.

[5] Miller D.A.: "A Theory of Modules for Logic Programming", *IEEE Symp. on Logic Programming*, Sept.1986, 106-114.

[6] Monteiro L., Porto A.: "Contextual Logic Programming", in *Proc. Sixth Int. Conf. of Logic Programming*, Lisbon, 1989.

[7] Lamma E., Mello P., Natali A.: "The design of an abstract machine for efficient implementation of contexts in Logic Programming", in *Proc. Sixth Int. Conf. of Logic Programming*, Lisbon, 1989.

[8] Clocksin W.F., Mellish, C.S.: *"Programming in Prolog"* (3rd Edition), Springer-Verlag, 1987.

[9] Moscowitz Y., Shapiro E.: "Static and Dynamic Semantics for Lexical Logic Programs", submitted for pubblication, 1989.

[10] Martelli A., Rossi G.F.: "On the semantics of Logic Programming Languages", in *Third Int. Conf. on Logic Programming - LNCS n.225*, Springer-Verlag, 1986, 327-334.

[11] Mello P., Natali A., Ruggieri C.: "Logic Programming in a Software Engineering Perspective", in *Logic Programming -Proc. of the North American Conference* (Lusk E.L. and Overbeek R.A., eds), The MIT Press, 1989.

[12] Martelli A., Rossi G.F.: "Enhancing Prolog to Suppot Prolog Programmning Environments", in *ESOP88 - LNCS 300* (Ganzinger, ed.), Springer-Verlag, 1988.

[13] Rossi G.F.: "Meta-programming facilities in an extended Prolog", in *Artificial Intelligence and Information-Control Systems of Robots* (I.Plander, ed.), North Holland, 1989.

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

[15] Brogi A., Lamma E., Mello P.: "Programming by composing logic open theories", Rapporto Interno DEIS Bologna, 1989.

[16] Miller D.A.:"Lexical Scoping as Universal Quantification", in *Proc. Sixth Int. Conf. of Logic Programming*, Lisbon, 1989.

APPENDICE - Esempi di programmi in Prolog strutturato

In questa Appendice sono raccolti un certo numero di semplici programmi logici (molti dei quali ripresi da [8]) scritti utilizzando il linguaggio con blocchi e moduli definito nei capitoli precedenti. Scopo di questa Appendice e' quello di mostrare che il linguaggio definito puo' costituire uno strumento di effettiva utilita' per la scrittura di programmi logici strutturati.

Esempio 1 - blocco semplice.

Predicato *sublist(L1,L2)*: vero quando *L1* e' una "sottolista" di *L2*.

```
sublist([X|L],{X|M]) :-
              {prefix([],_).
               prefix([X|L],[X|M] :- prefix(L,M)} =>
          prefix(L,M),!.
sublist(L,[_|M]) :- sublist(L,M).
```

Esempio 2 - blocchi annidati.

Predicato *eqset(X,Y)*: definisce l'uguaglianza tra due insiemi X e Y rappresentati come liste.

```
eqset(X,X) :-!.
eqset(X,Y) :-
              {eqlist([],[]).
               eqlist([X|L1],L2):-
```

```
                    localvars L3:
                          {delete([Z|Y],Y) :- Z=X.
                           delete([Y|L1],[Y|L2) :- delete(L1,L2)} =>
                    delete(L2,L3),
                    eqlist(L1,L3)} =>
        eqlist(X,Y).
```

In questo esempio, il predicato *delete* e' definito in un blocco interno al predicato *eqlist* che, a sua volta, e' definito in un blocco interno al predicato *eqset*. Da notare anche che la variabile X nella definizione del predicato *delete* e' usata come variabile globale.

Esempio 3 - congiunzione di goal nel "body" di un blocco.

Predicato *bsort(L,S)*: *S* e' la lista *L* ordinata secondo l'algoritmo di "bubble sort".

```
bsort(L,S) :-
        localvars M:
           (localvars [A,B,X,Y]:
                  {append([],L,L).
                   append([X|L1],L2,[X|L3]):-append(L1,L2,L3)} =>
              (append(X,[A,B|Y],L),
                  {order(...).
                   order(...). ...} =>
               order(B,A),!,
               append(X,[B,A|Y],M)) ),
        bsort(M,S).
bsort(L,L).
```

In questo esempio si nota che il "body" di un blocco puo' essere anche una ***congiunzione di goal*** e non soltanto un singolo goal come negli esempi precedenti. Inoltre, un goal del "body" puo' essere a sua volta un goal blocco permettendo cosi' di avere anche in questo caso blocchi annidati. Nell'esempio, il predicato *order* e' definito in un blocco interno al "body" del blocco contenente la definizione di *append*. Da notare anche che si e' scelto di definire la variabile M come locale a tutto il corpo della prima clausola del predicato *bsort* mentre le variabili A, B, X e Y sono locali al blocco contenente la definizione di *append*. La definizione del predicato *order* potrebbe essere data anche nell'ambiente piu' esterno, cioe' nell'ambiente contenente il blocco che definisce il predicato *append*, piuttosto che come blocco interno al predicato *append* stesso. Infatti, la definizione di *order* sarebbe comunque visibile dall'interno del blocco dell'*append* grazie alle regole di scope per le clausole previste dal linguaggio.

Esempio 4 - procedure.

Come esempio d'uso del costrutto delle procedure introdotto nel cap. 3, mostriamo la definizione di un predicato *split(In,Odd,Even)* che permette di suddividere una lista di numeri interi *In* in due sottoliste *Odd* e *Even* conteneti rispettivamente gli elementi pari e dispari di *In*. L'esempio e' ripreso da [9] e risulta particolarmente interessante proprio se confronto con quello riportato in [9]. Il confronto evidenzia una maggior leggibilita' della nostra soluzione.

```
split(In,Odd,Even) isproc
      {odd(X) isproc {.....}.
       even(X) isproc {.....}.
       split([X|In1],Odd1,Even1) :-
                  {split1 :- odd(X),Odd=[X|Odd1],Even1=Even.
                   split1 :- even(X),Even=[X|Even1],Odd1=Odd} =>
             split1,
             split(In1,Odd1,Even1).
       split([],[],[])}.
```


I predicati *odd* e *even* sono definiti come procedure locali alla procedura *split*. Il predicato *split1* invece fa uso di diverse variabili globali per cui e' conveniente sia definito come blocco interno alla *split*.

Esempio 5 - moduli parametrici.

In questo esempio ridefiniamo il modulo *lists*, gia' utilizzato negli esempi del capitolo 4, supponendo che le diverse implementazioni della procedura di ordinamento di una lista siano tutte raccolte in un unico modulo parametrico (rispetto al tipo di elementi della lista e alla relazione di ordinamento adottata), denominato *sort*. Il modulo *lists* semplicemente esporta una procedura *sort(L1,L2)* che al suo interno richiama uno dei diversi algoritmi di ordinamento previsti dal modulo *sort*, con una certa relazione di ordinamento, cosi' come specificato dai due parametri *Sortalg* e *Rel*, rispettivamente. La variabile individuale T e' stata introdotta per semplificare l'uso del modulo *lists*, dal momento che essa permette all'utente di ignorare i nomi dei moduli che definiscono le diverse relazioni di ordinamento per i diversi tipi di elementi della lista, permettendogli di specificare semplicemente tali tipi con le costanti *int*, *char*, ecc. Da notare che T nelle clausole di *sort* e' usata come variabile globale.

```
lists(Sortalg/2,T,Rel)  ismod
      {member(X,L) isproc   {...}.
       append(L1,L2,L3) isproc     {...}.
       ...
       sort(L1,L2) isproc
            {sort(L1,L2) :- T=int,!, demo    (sort(intord,Rel),Sortalg(L1,L2).
             sort(L1,L2) :- T=char,!, demo    (sort(charord,Rel),Sortalg(L1,L2). ...}.
       ... }.
sort(Ord,Rel/2) ismod
      {quicksort(L1,L2) isproc
            {quicksort([],[]).
             quicksort([H|T],S):-localvars     [A,B,A1,B1]:
                  split(H,T,A,B),
                  quicksort(A,A1),quicksort(B,B1),
                  demo(lists,append(A1,[H|B1],S))}.
      bsort(L1,L2) isproc     {...}.                    (ved. Esempio 3 - Appendice)
       ... }
      body
      {split(H,[A|X],[A|Y],Z):-
            demo(Ord,Rel(A,H)),split(H,X,Y,Z).
       split(H,[A|X],Y,[A|Z]):-
            demo(Ord,Rel(H,A)),split(H,X,Y,Z).
       split(_,[],[],[]).
       ... }.
intord ismod
      {gt(X,Y) :- ...
       lss(X,Y) :- ...
       equal(X,Y) :- ...}.
  ...

?- demo(lists(quicksort,int,gt),sort([3,1,2],Sorted)).
Sorted = [1,2,3]
?- demo(lists(bsort,char,lss),sort([b,c,a],Sorted).
Sorted = [c,b,a]
```

NEGAZIONE COME INCONSISTENZA E MECCANISMI DI STRUTTURAZIONE PROLOG

C. Bassino, C. Charrere, B. Demo

Dipartimento di Informatica, Universita' di Torino
c. Svizzera 185 - 10149 TORINO (ITALY)
tel. + (39)(11)7712002

1. INTRODUZIONE

I problemi inerenti la negazione per fallimento sono ampiamente discussi nella letteratura [AptB86, Clark78, Lloyd87, Shep87]. Le proposte per superarli riguardano da una parte restrizioni delle classi di programmi, attraverso per esempio le varie proprieta' di stratificazione, dall'altra forme differenti di negazione, quali la negazione come inconsistenza [Gabbay86] o la negazione costruttiva [Chan88, FooR88].
Nel proporre la negazione come inconsistenza, l'idea di Gabbay e Sergot e' quella di avere un programma logico dove specificare sia la conoscenza positiva su una realta', quale e' un normale programma Prolog, sia la conoscenza su quanto e' falso della medesima con formule negative non ammesse in Prolog. *Aspetto importante della proposta e' che le due componenti di descrizione della realta' ed il loro uso possono essere ricondotti ad un contesto Prolog* esteso con particolari meccanismi di strutturazione o, equivalentemente, esteso con i meccanismi per il ragionamento ipotetico descritti in [Gabbay85].
Questo lavoro si concentra sui meccanismi di strutturazione ed e' articolato in due parti. Nella prima e' presentata la negazione come inconsistenza e, attraverso degli esempi, sono evidenziate le relazioni tra tale negazione e meccanismi di strutturazione Prolog, quali moduli e blocchi, dove sia prevista una legatura dinamica delle variabili di modulo (modulo parametrico). Nella seconda parte si delinea come la negazione per inconsistenza possa essere realizzata in Alpes-Prolog, proposta di strutturazione di Prolog descritta in [Mello88, Mello89]. Per una piu' ampia descrizione della realizzazione si rimanda a [Bassi89].

2. NEGAZIONE COME INCONSISTENZA

2.1 Definizione

Come abbiamo accennato, nella negazione per inconsistenza un programma ha due componenti: un insieme di formule P che definiscono cio' che si vuole abbia successo ed un insieme di formule N che definiscono cio' che si vuole non abbia successo, quindi un programma e' una coppia (P, N).
I goal sono goal normali nel senso di [Lloyd87], cioe' formule composte da letterali con congiunzione e negazione, eventualmente annidate.
In [Gabbay86], la negazione per inconsistenza usando il programma (P, N) e' cosi' definita:

not G ha successo da (P, N) se $(P \cup \{G\}, N)$ *e' inconsistente.*

E, quindi, "si vede subito che si puo' assumere come definizione, tra quelle equivalenti" [Lolli87],

not G ha successo da (P, N) se un n $\in$ *N e' derivabile da P* $\cup$ *{G}.*

Questa seconda definizione fa intuire che e' possibile, con le opportune precisazioni, ricondurre le due componenti in cui e' descritta la realta' ed il loro uso nel valutare risposte, ad un contesto a clausole di Horn con la risoluzione come regola di calcolo e, quindi ad un contesto Prolog.
La questione va pero' opportunamente precisata. Seguendo la originiaria presentazione di Gabbay e Sergot, ne introduciamo i vari aspetti per gradi, considerando prima il caso senza variabili e poi il caso del primo ordine. Come vedremo, la negazione come inconsistenza e' ricondotta ad un ambito Prolog con una strutturazione del programma, cioe' usando costrutti quali blocchi o moduli. Piu' precisamente, oggetto del presente lavoro e' la necessita' di usare il costrutto di modulo parametrico, tipo quello previsto in Alpes-Prolog [Mello88], messa in evidenza attraverso esempi.

Si noti che, dal punto di vista logico, la negazione come inconsistenza non pone problemi poiche' e' una negazione logica: in particolare, non c'e' problema di completezza e si ha monotonicita'. Quanto ai rapporti con la negazione come fallimento, questa e' un caso particolare della negazione per inconsistenza [Gabbay86].

2.2. Caso Proposizionale

Consideriamo un primo esempio G1 di goal.

G1= not(d $\wedge$ a).

Dalla definizione su data: not(d $\wedge$ a) ha successo da un programma (P, N) se un n $\in$ N e' derivabile da P' = P $\cup$ { d $\wedge$ a }.
La valutazione di G1 avviene dunque mediante la verifica di inconsistenza di un insieme di clausole P' ottenuto da quello di partenza P esteso con l'insieme di clausole {d $\wedge$ a}.
La verifica dell'inconsistenza di P' potrebbe comportare a sua volta la valutazione di un ulteriore goal negativo not G1'. Sarebbe necessario, allora, dimostrare l'inconsistenza di un programma nuovamente esteso con un secondo insieme di clausole P' $\cup$ { G1' }.

Analogamente, varie estensioni del programma (P, N) sono necessarie se si hanno da valutare goal con negazioni annidate.

Negazioni annidate. Se la definizione di negazione per inconsistenza dice che dato il programma (P, N)
not G ha successo da (P, N) se (P $\cup$ *{G}, N) e' inconsistente,*
per il caso delle negazioni annidate si avra'
not (G1 $\wedge$ *not G2)) ha successo da (P, N) se (P* $\cup$ *{G1}, N* $\cup$ *{G2}) e' inconsistente.*

Si consideri il seguente esempio: siano dati il programma iniziale (P,N), con N vuoto, ed il goal

G2= not (d $\wedge$ a $\wedge$ not (b $\wedge$ not c))

G2 ha successo dal programma (P,N) se il programma :

(P',N') = (P $\cup$ {a, d}, N $\cup$ {(b $\wedge$ not c)})

risulta inconsistente. Per dimostrare l'inconsistenza del programma corrente (P', N'), deve essere valutato il goal (b $\wedge$ not c) unico elemento di N'. Tale goal ha successo se il programma

(P'', N'') = (P $\cup$ { a, b } $\cup$ { b } , N $\cup$ { c })

e' inconsistente. Il passo successivo della computazione consistera' dunque nel verificare se in N'' vi e' un goal che ha successo da P''.

Quello che e' importante notare e' che la verifica del goal iniziale G2 a partire dal programma (P, N) ha come conseguenza il considerare diversi programmi (P', N'), (P'', N'') creati dal programma (P, N) esteso con insiemi di clausole differenti, quando si deve valutare un goal negativo. Per ogni programma viene fatta la prova della inconsistenza di un programma ottenuto estendendo il programma corrente con un insieme di clausole.

Unita' distinguibili. E' importante osservare che le clausole che vengono aggiunte ad un programma devono poter essere distinte dalle clausole del programma esteso, cioe' costituire un blocco o unita' di clausole identificabile, poiche', al termine della verifica dell'inconsistenza del programma corrente, si deve poter eliminare l'unita' aggiunta e continuare la computazione col programma precedente. L'esempio che segue illustra questa esigenza. Dato il goal

G3 = not (d) $\wedge$ not (c $\wedge$ not a)

si vuole dimostrare la validita' di G3 a partire dal programma iniziale (P,N). Il goal G3 e' costituito dalla congiunzione dei due sottogoal negativi G3.1= not (d) e G3.2= not (c $\wedge$ not a). La valutazione di G3 dovra' dunque essere la congiunzione dei risultati della valutazione dei due sottogoals.
La valutazione del goal G3.1 consiste nel verificare l'inconsistenza del programma:

(P',N') = (P $\cup$ {d}, N)

Se not(d) ha successo, cioe' se esiste un n in N' che ha successo da P', procediamo a valutare G3.2. Tale valutazione e' compiuta sul programma (P,N) iniziale; da (P',N') vanno percio' eliminate le clausole che sono state aggiunte per dimostrare il goal G3.1. G3.2 risulta verificato se il programma seguente

(P'',N'') = (P $\cup$ {c} , N $\cup$ { a })

risulta consistente.

L'esempio che segue integra le considerazioni fatte finora. Dato il goal

G4 = not (a $\wedge$ b $\wedge$ not(c $\wedge$ not(d $\wedge$ not e)) $\wedge$ not f)

e un programma iniziale P= {c}, i blocchi considerati durante la computazione sono i seguenti :

blocco 1 = { c }
blocco 2 = { a, b, not(c $\wedge$ not(d $\wedge$ not e)), not f }
blocco 3 = { d, not e }

All'inizio della computazione ci troviamo a livello 1 e l'unico blocco attivo e' il blocco 1.

La valutazione del goal G4 comporta la creazione del blocco 2 e la verifica, a livello 2, dell'inconsistenza del programma dato dalla unione dei blocchi 1 e 2. Si prova a dimostare il goal c ∧ not (d ∧ not e). Il fatto c ha immediatamente successo. Il goal not (d ∧ not e) provoca invece l'allocazione del blocco 3 e il passaggio ad un nuovo livello, il terzo, in cui bisogna provare l'inconsistenza del programma formato dai blocchi 1, 2 e 3 .Tale programma com'e' facilmente verificabile, risulta consistente. Si scende allora al livello precedente, escludendo dal programma il blocco 3 e concludendo che il goal c ∧ not (d ∧ not e) non ha successo e quindi non determina l'inconsistenza del programma formato dai blocchi 1 e 2. Si tenta di verificare l'inconsistenza lanciando un altro goal negativo, f, ma anche questo non ha successo. Poiche' non vi sono altri goal negativi, si torna al livello precedente che corrisponde al primo, escludendo il blocco 2, e si conclude che il goal iniziale G4 non ha successo a partire dal programma iniziale ({c}, {}).

L'idea generale e', quindi, quella di risolvere un goal del tipo not (a ∧ not c) aggiungendo la clausole "a" e "not c" alle clausole appartenenti agli eventuali blocchi piu' esterni (P, N) che contengono il goal dato.
L'ambiente di computazione si modifica in una forma ad albero, dove la radice e' il programma originario esteso con unita' differenti (rami di livello 1), questi a loro volta eventualmente estesi e cosi' via.
Si noti che la configurazione ad albero non e' soltanto propria della valutazione di un goal complesso cioe' con piu' negazioni ed annidamenti di negazioni, come visto negli esempi bensi' si ripropone quando si debba, durante la valutazione di goal negato, valutare nel corpo di una clausola un letterale negativo.

Prima di passare a descrivere una possibile realizzazione della negazione per inconsistenza, consideriamo ancora un esempio che coinvolge delle variabili.

2.3 Un Caso del Primo Ordine

La presenza di variabili nel programma e nel goal da valutare determina degli aspetti nuovi da considerare durante la valutazione: quelli derivanti dalla quantificazione esistenziale del goal medesimo.
Siano dati il programma

P = {b(a,T):-a(T) }
N = {}

e il goal G = not (a(Y) ∧ not b(Y,Z)).
Il goal e' quantificato esistenzialmente, quindi si devono cercare un y0 e uno z0 tale che il goal not (a(y0) ∧ not b(y0,z0)) abbia successo. Per risolvere questo goal, e' necessario aggiungere a (P, N) la clausola ∃ (Y,Z) (a(Y) ∧ not b(Y,Z)) e verificare l'inconsistenza del programma P' ottenuto.

P' = {b(a,T):-a(T)
∃ Y a(Y) }
N' = {b(Y,Z) }

Tutte le sostituzioni di variabili eseguite nella clausola ∃ (Y,Z) (a(Y) ∧ not b(Y,Z)), durante la verifica di inconsistenza devono essere persistenti, cioe' una volta che una variabile viene sostituita con un certo valore, essa mantiene tale valore fino alla fine della computazione. Inoltre la variabile Y del fatto a(Y) e' legata alla variabile Y della clausola b(Y,Z) in N' poiche' a(Y) e not b(Y,Z) sono componenti della stessa clausola originaria ∃ (Y,Z) (a(Y) ∧ not b(Y,Z)).
Per valutare l'inconsistenza del programma P', bisogna valutare il goal b(Y,Z) unico elemento di N'. Questo fatto unifica con la testa della clausola b(a,T):-a(T) in P' attraverso la sostituzione t = { T/Z, Y/a }. A questo punto la teoria (P', N') diventa la teoria (P', N'){ T/Z, Y/a } cioe'

P'{T/Z, Y/a} = {b(a,Z):-a(Z)
a(a) }
N'{T/Z, Y/a} = { b(a,Z) }

su cui dobbiamo valutare il goal a(Z) e questo ha successo per Z = a.

Nella verifica di inconsistenza si deve dunque tener conto che (1) gli elementi che vengono aggiunti al programma per la valutazione di un goal sono quantificati esistenzialmente e le sostituzioni delle variabili di tali elementi devono essere persistenti, (2) deve esistere un meccanismo tale da garantire che una variabile di uno stesso goal anche se il goal compare parte nella componente positiva P, parte in quella negativa N, sia sempre legata nello stesso modo.

Per mezzo di esempi abbiamo fin qui messo in evidenza le caratteristiche piu' rilevanti della negazione per inconsistenza di cui si deve tener conto in una realizzazione che riconduca la proposta ad contesto Prolog. Riassumendole, cominciamo con una osservazione sulla rappresentazione della coppia (P, N) che va integrata in un unico programma Prolog. Le clausole di P sono clausole di Horn estese con sottogoal negati per inconsistenza quindi possono costituire un programma Prolog H. Quanto all'insieme di clausole negate elementi di N, una soluzione, molto adottata anche in altri contesti, e' quella di averle espresse nel medesimo H ciascuna come corpo di una clausola avente l'atomo "vinc" come testa. Ad esempio, la clausola not a si esprimera' con la clausola vinc:-a.
La testa "vinc" comune distingue queste clausole in H come quelle in N nella teoria (P, N).

Quanto alla valutazione di un goal sul programma H esprimente la teoria (P, N), gli esempi mostrano che la teoria considerata in ogni momento della computazione e' data da un insieme di blocchi di clausole. All'inizio della computazione, la teoria originaria puo' essere vista come il blocco di livello 1. La valutazione di un goal negativo richiedera' di andare a considerare teorie ottenute da piu' blocchi di clausole.
E' da notare che il programma iniziale viene considerato come il blocco piu' esterno e che le clausole del goal aggiunte vengono rimosse non appena la dimostrazione dell'inconsistenza del programma ottenuto e' terminata. Notiamo, inoltre, che l'aggiunta di un insieme D di clausole ad un programma P comporta che tutte le clausole di D siano visibili da P e viceversa. La dimostrazione di un goal not G comporta la verifica