PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A PROGRESS REPORT ON THE LML PROJECT

Bruno Bertolino, Paolo Mancarella, Luigi Meo, Luca Nini, Dino Pedreschi, Franco Turini

Dipartimento di Informatica, Università di Pisa
Corso Italia, 40, I - 56100 Pisa, ITALY

ABSTRACT

LML is a programming language for the construction of knowledge based systems. Chunks of knowledge are represented as logic programming theories. A functional metalevel provides suitable mechanisms for the dynamic handling of logic theories. From an application viewpoint, the possibility of a clean handling of the dynamic evolution of knowledge over time as well as of a modular approach to knowledge based systems development is the most distinctive feature of the language. The paper presents the language, its implementation and a consistent example of its use.

1 INTRODUCTION

It is, nowadays, widely recognized that Logic Programming (Kowalski 1979, Kowalski 1983) is one of the best basis for constructing knowledge based systems (Gallaire 1987). The consequences are a great push towards the study of general techniques amenable to this endeavor. Among them we recall various attempts of making the basic features of Logic Programming executors, i.e. unification and backtracking, more powerful, the idea of metaprogramming and metainterpretation (Bowen and Kowalski 1982), the study of techniques for structuring logic programs and the integration of Logic Programming with other paradigms (Darlington et al. 1985, Robinson and Sibert 1982).

The project described in this paper, named LML for Logical ML, is concerned in a way or another with the last three of the above topics.

In a few words, LML is a language for building knowledge based systems, in which logic theories can be manipulated, as values of an ordinary data type, by a functional meta-layer, which is essentially Standard ML (Milner 1985).

The distinguishing feature of LML with respect to other proposals aimed at the modularization of Logic Programs, is the possibility of manipulating logic theories at run time in an *intensional* way. Essentially, the functional layer embodies a number of operators capable of transforming one or more logic theories into a new one (Mancarella et al. 1988b).

It is well known that one of the toughest problems in dealing with knowledge based systems is handling the evolution over time of the knowledge (e.g. learning, nonmonotonic reasoning etc.). Our hope and intents are that LML can be a clean and flexible solution to this class of problems.

The paper is organized as follows. Section 2 presents the most distinguishing features of the LML language, providing also some small examples. Section 3 offers a rather complex example, which, in our opinion, is necessary to provide a sharper flavor of the expressiveness of the language. Section 4 discusses the current implementation of the language, stressing the most critical points. At last, section 5 concludes the paper by outlining the theoretical foundations of LML.

2 OVERVIEW OF LML

2.1. The functional kernel of LML

The functional language which provides the meta-level of LML is essentially the kernel of Standard ML (Milner 1985) extended with *non-strict* functions. 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*. 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"

```
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. Since also type constructors are functions, it is possible to define and use potentially infinite data structures. The following definitions are examples of this feature:

```
val  Ones = 1 :: Ones;
val  From n = n :: From (n +1);
val  Numbers = From 0;
```

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

```
val  First 1 (a::R) = (a)   |
     First n (a::R) = (a :: First n -1 R)
val  FirstTwenty = First 20
```

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 are hard to manage in strict functional programming languages (Henderson 1980, Richards 1982).

2.2. Definition and intensional manipulation of logic theories

The logic component of LML essentially provides logic theories as data types, along with a suitable set of operators for manipulating them. Logic theories are defined within an extension of pure Horn Clause Logic, which allows the use of both *negation* and *universal quantification* in clause bodies. The formal semantics of LML, and in particular the model theoretic semantics of its logic component, will not be discussed in this paper. Just as a hint, the problem of assigning a suitable meaning to a logic program T where negation and universal quantification are used freely in its clause bodies, is approached taking into account the first order approximation of two extra axioms, namely the *Closed World Assumption* (Reiter 1978) and the *Closed Domain Assumption* (Lloyd and Topor 1985, Lloyd and Topor 1986, Mancarella et al. 1988a). The approximation of CWA consists in considering the *completion* of T (Lloyd 1984, Shepherdson 1985), whereas the approximation to CDA consists in imposing type constraints on T. This approach, discussed deeply in (Barbuti et al.1988), is inherited from similar machineries used in (Lloyd 1984) for general logic programs and in (Lloyd and Topor 1985, Lloyd and Topor 1986) for deductive databases.

Logic theories deal with the same (concrete) data types which are definable in the functional component. As a consequence, clauses are typed just in the same sense functions are. However, for the sake of simplicity, from now on we will assume a single type for clauses, disregarding, henceforth, type information in the definitions. More precisely, the definitions will be stated with respect to a generic type τ equipped with constructors $c_1,\ldots,c_k$ each of which of arity $a_i \geq 0$.

Definition [clauses]
A clause has the following form

$$p(t_1,\ldots,t_n) :- B_1,\ldots,B_k$$

where $(t_1,\ldots,t_n)$ is an *unrestricted* tuple of terms (i.e. each variable occurs exactly once in it).
Each B_i has one of the following forms

- L (a literal)
- **exists** z.L
- **all** z.L

where z are the variables occurring only in L.
Literals are

- $q(t_1,\ldots,t_n)$ (positive literal)
- $q\sim(t_1,\ldots,t_n)$ (negative literal)

The explicit mention of the variables bound by **exists** and **all** will be omitted whenever the meaning is clear from the context. The condition on unrestrictness on the terms in the clause heads does not affect generality, since equality constraints can be explicitely stated in the body via an equality predicate (Barbuti et al. 1987, Barbuti et al. 1988).

Definition [basic theory]
A basic theory is either the EmptyTheory, defining no predicate, or a clause or the juxtapposition – denoted by ";" – of two basic theories

As an example, consider the following theory defining the *ancestor* relation.

```
val Ancestors =
    {ancestor (x,y) :-  x=y ;
```

```
ancestor (x,y) :-
    exists z.proper_ancestor(x,y,z) ;
proper_ancestor(x,y,z) :-
    parent(x,z),ancestor(z,y)}
```

Notice the possibility of denoting a theory with an identifier (Ancestors in the example) as for any other object of the language. The equality predicate = in the first clause is used in order to make its head unrestricted.

Besides the juxtapposition on basic theories, a dual operator, namely the *meet* operator, is provided.

Definition [meet]
Given two basic theories T_1 and T_2, for each predicate name p, if

$p(t_1,\ldots,t_n)$:- B_1 is a clause of T_1 and
$p(t'_1,\ldots,t'_n)$:- B_2 is a clause of T_2 and
it does exist θ = m.g.u.($(t_1,\ldots,t_n)$, $(t'_1,\ldots,t'_n)$)
then
$p((t_1,\ldots,t_n)\theta)$:- $B_1\theta, B_2\theta$ is a clause of meet(T_1, T_2)

Actually, the theories of LML have two separate components:

- a *positive component* (denoted by T^+), i.e. a basic theory, providing the clauses for positive literals;
- a *negative component* (denoted by T^-), i.e. a basic theory providing the clauses for negative literals.

In fact, the negative information is handled considering, for each predicate p, a new predicate symbol p~, the definition of which embeds the negative knowledge of p. The negative component is, indeed, obtained via a transformation, called *intensional negation*, applied to the positive one. The transformation is an extension of the techniques introduced by Sato and Tamaki (1984).

2.3. Intensional Negation

The first step in the definition of intensional negation is the algorithm for negating a tuple of terms. This algorithm mirrors the idea that a term (and equivalently a tuple of terms) represents the set of its ground instances. The negation algorithm yields an intensional representation of the complement of this set by means of a set of terms. The algorithm works properly only if applied to unrestricted tuples of terms, and furthermore it yields a *finite* set of tuples. It has been shown by Lassez and Marriot (1987) that such a finite representation does not exist for restricted tuples.

Definition [negation of terms]
Referring to the type τ equipped with constructors $c_1,\ldots,c_k$ each of arity $a_i \geq 0$, given an unrestricted term t, negt(t) denotes its negation and it is computed as follows
(1) negt(x) = { }
(2) negt($c_i(t_1, \ldots, t_{a_i})$) =

$\{c_j(x_1,\ldots,x_{a_j}) \mid j \neq i\} \cup$
$\{c_i(x_1,\ldots,x_{k-1}, t', x_{k+1},\ldots,x_{a_i}) \mid k \in [1,a_i], t' \in \text{neg}(t_k)\}$

where all the newly introduced variables are fresh and distinct.

Notice that the second rule collapses into
negt(c_i) = $\{c_j(x_1,\ldots,x_{a_j}) \mid j \neq i\}$
when c_i is a nullary function, i.e. a constant. The negation algorithm can be easily extended to tuple of terms in the following way:

Definition [negation of tuples of terms]
neg $(t_1,\ldots,t_n)$ =
$\{ (x_1,\ldots,x_{k-1}, t', x_{k+1},\ldots,x_n) \mid k \in [1,n], t' \in \text{neg}(t_k) \}$

As an example, the instantiation of the negation algorithm on the data type of natural numbers, with the constant zero and the unary function succ, is the following
neg(x) = { }
neg(zero) = {succ(x)}
neg(succ(t)) = {zero} $\cup$ {succ(t') | t' $\in$ neg(t)}
which, applied to the term succ(succ(zero)) yields
{zero, succ(zero), succ(succ(succ(x)))}

We are now ready to define the intensional negation of a basic theory, which defines only the positive predicates. In order to simplify the notation the lower case letter n, possibly with subscripts, will be used in the sequel to denote tuples of terms.

Definition [intensional negation]
Let T^+ be a non empty basic theory. The intensional negation T^- of T^+ is inductively defined as follows

(1) $(p(n)$:- $B_1,\ldots,B_k)^-$ =
$q\sim_1(x)$:- ;
.....
$q\sim_r(x)$:-
$p\sim(n_1)$:- ;
.....
$p\sim(n_h)$:- ;
$p\sim(n)$:- $B\sim_1$;
.....
$p\sim(n)$:- $B\sim_k$;
where
- $\{n_1,\ldots,n_h\}$=neg(n)
- $B\sim_i$ = $L\sim$ if $B_i = L$
  all z. $L\sim$ if B_i = **exists** z. L
  exists z. $L\sim$ if B_i = **all** z. L
  where
  $L\sim$ = $q\sim(s)$ if $L = q(s)$
  $q(s)$ if $L = q\sim(s)$
- $q\sim_i(x)$:- are introduced for each predicate symbol $q_i \neq p$, occurring in the body of the clause

(2) $(T_1;T_2)^-$ = meet(T_1^-, T_2^-).

Case 1 corresponds to the program consisting of a single clause p(n):-B_1,...,B_k: recalling that intensional negation allows to compute what can be inferred by *negation as failure*, let us notice that a goal q(m) fails iff either q(m) does not unify with p(n) (first h+r facts) or p(m) unifies with p(n) via mgu θ but $B_i\theta$ fails for some i (last k clauses). Case 2 finally states a kind of De Morgan rule for failure.

As an example, consider the following basic theory defining the predicate *even* on natural numbers:

```
T+:  even(zero);
     even(succ(x)) :- even~(x)
T-:  meet((even(zero) :-)-,
      (even(succ(x)) :- even~(x))-) =
     meet(  (even~(succ(x)) :-),
            (even~(zero) :-;
            even~(succ(y)) :- even(y))  ) =
            even~(succ(y)) :- even(y).
```

The whole theory is then:

```
even(zero);
even(succ(x)) :- even~(x)
even~(succ(y)) :- even(y)
```

which, as expected, defines even and odd (even~) numbers.

2.4. Intensional operators on logic theories

As mentioned in the introduction, the main goal of LML is to provide means for manipulating chunks of knowledge, coded as logic theories. Henceforth, a number of operators on theories is provided: renaming, negation, union and intersection.

Definition [renaming]
Given a theory T=<T+, T-> and $p_1,\ldots,p_m$, $p'_1,\ldots,p'_m$ predicate names, then

$T[p'_1/p_1,\ldots, p'_m/p_m]$

is the theory where each p_i is substituted by p'_i.

Definition [negation]
Given a theory T and a predicate name p, then the negation of T with respect to p is

not(T,p) = T [p/p~, p~/p].

As an example, consider

```
T+ = {p(succ(x)) :- exists y. r(y) ; r(zero):- }.
```

Then

```
T- = { p~(zero):- ;
       p~(succ(x)) :- all y. r~(y) ; r~(succ(x)) }.
```

Referring to the previous definitions:

```
not(T, p)+ = {p(zero):- ;
              p(succ(x)) :- all y. r~(y) ; r(zero) }
not(T, p)- = { p~(succ(x)) :- exists y. r(y) ;
              r~(succ(x)):- }.
```

The definitions of the union and the intersection of two theories are stated with respect to subtheories occurring in both theories, i.e. the set of clauses defining a specific predicate (maybe in different ways). If T is a theory and p is a predicate name, T↓p denotes such a subtheory, whereas T↑p denotes the remaining part.

Definition [union and intersection]
Given two theories T_1 and T_2 and a predicate name p, let

$R_1 = T_1[q'/q]$ and $R_2 = T_2[q''/q]$

for each predicate name q≠p occurring in both T_1 and T_2, with q' and q'' new predicate symbols. Then

union(T_1, T_2, p) =
$< (R_1^+; R_2^+), (R_1^-\uparrow p; R_2^-\uparrow p; meet(R_1^-\downarrow p; R_2^-\downarrow p)) >$
intersection(T_1, T_2, p) =
$< (R_1^+\uparrow p; R_2^+\uparrow p; meet(R_1^+\downarrow p; R_2^+\downarrow p)) , (R_1^-; R_2^-) >$

Notice that in both operations, if p is not defined in one of the two theories, it is enough to add the clause p~(x):- in it before the application of the operations. The extension of the operations to an arbitrary number of predicates is straightforward.

Finally, as a matter of syntactic sugar, the following abbreviations are available:

not T = not (T, <$p_1,\ldots,p_n$>)

where $p_1,\ldots,p_n$ are all the predicates occurring in T;

union(T_1,T_2) = union(T_1,T_2, <$p_1,\ldots,p_n$>)
intersection(T_1,T_2) = intersection(T_1,T_2, <$p_1,\ldots,p_n$>)

where $p_1,\ldots,p_n$ are all the predicates occurring in both T_1 and T_2.

Informally, putting theories together by union yields a new theory in which the original ones cooperate during deductions: in fact, Pedreschi (1988), Mancarella and Pedreschi (1988) show that the success set of union(T_1, T_2) includes the (set-theoretic) union of the separate success sets. In a dual way, the success set of intersection(T_1, T_2) is included in the (set-theoretic) intersection of the separate success sets.The following simple example points out these observations.

```
T1+= { p(succ(x)) :- q(x)  ;  q(succ(succ(zero))) :- }
T2+= { p(zero) :-  ;  p(succ(succ(x))) :- p(x) }.
```

Then

```
union(T1, T2, p)+ =  {
                       p(succ(x)) :- q(x) ;
                       q(succ(succ(zero))) :- ;
                       p(zero) :- ;
                       p(succ(succ(x))) :- p(x)
                       }
union(T1, T2, p)- =  {
                       q~(zero) :- ;
                       q~(succ(zero)) :- ;
                       q~(succ(succ(succ(x)))) :- ;
                       p~(succ(zero)) :- q~(zero) ;
                       p~(succ(succ(x))) :-
```

```
                    p~(x), q~(succ(x))
                }
```

Notice that in T_1 p(n) holds when n=3, whereas in T_2 p(n) holds on even numbers. Putting T_1 and T_2 together with respect to predicate p, p(n) holds when n≥3 or n is even, while p~(n) holds only if n=1.

On the other hand,

intersection(T_1,T_2,p)$^+$ =

```
        {p(succ(succ(x))) :- p(x), q(succ(x)) ;
         q(succ(succ(zero))) :-  }
```

intersection(T_1,T_2,p)$^-$ =

```
        {p~(zero) :- ;
         p~(succ(zero) :- ;
         p~(succ(succ(x))) :- p~(x) ;
         p~(succ(x)) :- q~(x) ;
         q~(zero) :- ;
         q~(succ(zero)) :- ;
         q~(succ(succ(succ(x)))) :-  }
```

Hence, in the intersection of the original theories with respect to p, p denotes the empty relation, and consequently p~(n) holds for each number n.

As a further example, consider again the Ancestors theory defined at the beginning of this section. In order to make it usable, it should be combined with some parents database, defining the predicate parent, e.g.

```
val ParentsDB={
          parent(Bill, Jane) :-   ;
          parent(Mary, Al) :-    ;
          ..............}
```

The combined theory

```
val  AncestorsDB =
        union(Ancestors, ParentsDB)
```

does the job. Notice that the intensional operators allow to model knowledge bases evolving over time, e.g. union can be used to augment theories by asserting new facts or rules, extending the behavior of Prolog built-ins like **assert**. When the fact that John is a child of Bob comes to evidence, then the `ParentsDB` theory can be updated accordingly as follows:

```
val  ParentsDB =
       union(ParentsDB, {parent(Bob,John)}).
```

2.5. Set expressions and extensional operators

As discussed in the introduction, set expressions are provided in order to denote sets, which are nothing but the set of answers obtained by evaluating a goal (i.e. a query) with respect to a theory.

For example the set expression

```
{(x,y)  | P(x,y,succ(succ(zero))) wrt Plus}
```

where

```
val
Plus = {P(x, zero, x) :- ;
        P(x, succ(y), succ(z)) :-
              P(x, y, z) }
```

evaluates to the set of pairs

```
{(x,y)  | x + y = 2} = {(0,2), (1,1), (2,0)}.
```

As a further example, referring to the AncestorDB theory of the previous section, the expression

```
{x | adam(x) wrt
    union({(adam(z) :- all y .ancestor(z, y) ),
          AncestorDB)}
```

yields the individuals which are ancestors of everybody, if they exists. Of course, if the ParentDB theory is well defined, such a set is either empty or a singleton.

Special set-expressions are

- *empty set* {}
- *extensional set* $\{E_1\ E_2 \ldots E_k\}$

Of course, set expressions are lazily evaluated, in the sense that the extension of a set is never computed unless explicitly requested by the application of an extensional operator. Nevertheless, set expressions can be intensionally manipulated through some suitable operators (union, intersection,...) corresponding to the analogous operators on theories. Their definitions are straightforward and we omit them for brevity. On the other hand, the following is a list of some extensional operators over sets which, whenever applied, cause the activation of the query evaluation process. SE stands for a set expression.

- **`for each x in SE do E`**
  evaluates to the set of values obtained by computing E with respect to each element of SE.
- **`all x in SE such that BE`**
  evaluates to the subset 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 such an element exists, otherwise it yields a failure.
- **`v isin SE`**
  evaluates to true if v is an element of SE.

The query evaluation process used in LML, called SLDN, is introduced in (Barbuti et al.1988, Mancarella 1988). SLDN is essentially SLD resolution augmented to deal with intensional negation. Since both the positive and the negative knowledge are explicitly expressed in an LML theory T=<T$^+$, T$^-$>, the evaluation of a negative literal p~(t) with respect to T consists simply of an SLDN refutation using the clauses for p~ in T$^-$. The only real extension of SLD is introduced to deal with universal quantified literals, of the kind **all** z.L. The refutation of such a goal exploits *negation as failure* as a way to check universally quantified theorems (Clark 1978, Lloyd 1984). For instance, the subgoal **all** y. q(x,y) is actually implemented as the conjunction q(x,y), *naf* q~(x,w). Roughly speaking, SLDN evaluates q(x,y) first, providing

candidate solutions $s_1,\ldots,s_k$ for x, while *naf* q~(x,w) discards those solutions s_i in correspondence to which the evaluation of q~(s_i, w) does not finitely fail. In other words, we use the failures of the predicate q~ to filter the success set of q with respect to the universally quantified variable. Using the failures of a negative predicate requires some care. The problem falls in the realm of so-called general programs, in which negation is freely used in a logic program. An example can help to illustrate the problem and the kind of solutions we are pursuing.

Consider the clause
p(x) :- q(x),r(x)
Our intensional negation algorithm yields the clauses
p~(x) :- q~(x);
p~(x) :- r~(x)

Such a program, although it works perfectly well for computing the negative information contained in p, is not logically equivalent to the logical negation of the original program, which is the **non** Horn clause

p~(x) :- q~(x) **or** r~(x)

In our system, the computation of such a clause is approximated, generating, for the negation of the clause p(x) :- q(x),r(x), either

p~(x) :- q~(x);
p~(x) :- q(x),r~(x)

if r recursively depends on p and q does not, or

p~(x) :- r~(x);
p~(x) :- r(x), q~(x)

if q recursively depends on p and r does not. If both r and q depends on p the negation is performed as usual. If the conditions hold, such a trick provides an effective way of computing the non Horn clause, providing the correct failure set for p~. The implementation of the technique obviously requires a static analysis of the program.

3 AN EXAMPLE

Game playing is a typical environment for AI applications, at least for exemplifying the expressiveness of AI formalisms. In this section we show (part of) a program which simulates a Cluedo's player, which has been used as a significant test case for our prototype system. Briefly, Cluedo is an investigation game, in which each player has some incomplete information about a crime and (s)he should identify the guilty person (there are six potential criminals), where the crime has been committed (there are six potential locations) and which weapon has been used (there are six possible weapons). Initially, three cards identifying the criminal, the weapon and the location are put in a box, and nobody can see them. Moreover, the remaining cards are distributed among the players. In this way, each player initially has just partial information about the crime, for instance if (s)he possesses the card identifying person `P`, then (s)he knows for sure that `P` is not guilty. Each player in turn may perform one of the following actions:

i) try an answer, namely a location, a weapon and a person;
ii) ask for three cards, a location, a person and a weapon.

In case i) if the answer is correct, the player wins and the game is over. If the answer is incorrect then the player is out of the game, in the sense that from now on (s)he should just answer to the others' requests. In case ii) the first player (within a predefined order) who possesses one of the three cards asked for, shows it to the enquiring person and only to him. So, in this way, the asking person increases its knowledge about the crime; moreover the other players' knowledge increases too, even if by hypothesis and not by certainity.

In Cluedo each player should:

a) maintain somehow its knowledge about the crime, which augments over time;
b) decide which question is a good one for case ii) above;
c) decide if the actual information is strong enough to determine the right answer.

LML provides the right mechanisms to simulate these three components of a player's behavior and, as we show in the rest of this section, each one can be handled at the right abstraction level.

First of all, let us define the data types identifying the game's components:

```
type Players = player of nat;
type Suspects = s1 | s2 | s3 | s4 | s5 | s6;
type Locations = l1 | l2 | l3 | l4 | l5 | l6;
type Weapons = w1 | w2 | w3 | w4 | w5 | w6;
type Cards = susp of Suspects |
      loc of Locations | weap of Weapons.
```

Let us assume, for the sake of simplicity, that the LML program we are writing corresponds to `player(1)`. The knowledge base of `player(1)` consists of several kind of facts which the player knows for certain and of several kind of hypotheses, typically corresponding to disjunctive information. The following is a list of some of the predicates corresponding to these kinds of knowledge, which are self-explanatory:

- `has(P, C)`
  `has_not(P, C)`
  `has_not_all_three(P, C1, C2, C3)`
- `has_not_both(P, C1, C2)`
  `has_at_least_one_of_three(P, C1, C2, C3)`
  where `P` is of type `Players` and `C, C1, C2, C3` are of type `Cards`.

First of all, notice that the predicate `has_not` must not be confused with the (intensional) negation `has~` of `has`. In fact, the intended meaning of `has_not(P, C)` is that we know *for certain* that player `P` has not got card `C`, which is not the same as `has~(P, C)`, which holds whenever, from the current (incomplete) knowledge base, we can not conclude `has(P, C)`. More formally, the following

implication, but not the converse one, holds

```
has_not(P, C) → has~(P, C)
```

The LML program simulating the Cluedo's player will augment the knowledge base by a fact of the kind `has_not(P,C)` for `C=C1`, `C=C2` and `C=C3` whenever a player asked for the triple `C1`, `C2`, `C3` and player `P` was able to show none of them.

The predicate `has_at_least_one_of_three` is an example of *disjunctive* knowledge. In fact, `player(1)` can infer the fact `has_at_least_one_of_three(P, C1, C2, C3)`, whenever another player asked for the triple `(C1, C2, C3)` and player `P` showed one of the three cards (we dont't know exactly which one) to the asking person.

Whenever `player(1)` must ask for a triple, some rule has to be applied in order to find out which cards *may be in the box*. This is achieved simply by the following definition of the predicate `request`

```
request(P, L, W) :-
   may_be_in_the_box(susp(P)),
   may_be_in_the_box(loc(L)),
   may_be_in_the_box(weap(W))
```

where

```
may_be_in_the_box(Card) :-
   all has~(P, Card),
   all is_better~(C, Card)
```

The definition of `is_better` is omitted here, but intuitively `is_better(C1, C2)` holds whenever the knowledge base implies that `C1` is in the box. Notice the use of `has~` instead of `has_not` in the above definition: the goal `all has~(P, Card)` succeeds when for each player `P`, the information in the knowledge base does not allow to state that player `P` has got card `Card` which means exactly that `Card` *may be in the box*. The following definition of the relation clarifies further this point

```
is_in_the_box(Card) :- all has_not(P, Card).
```

We can now define the logic theory `choice` which embeds the above definitions of the predicates involved in determining either the solution or a good request:

```
theory choice =
        {request(P, L, W) :- .....
         may_be_in_the_box(Card) :- .....
         is_better(C, Card) :- .....
         is_in_the_box(C) :- .....}
```

A separate theory, named `infer`, is defined, containing the deduction rules which allow to infer facts from the knowledge base. The following are some of the definitions included in such a theory

```
theory infer =
{has_not_both(P, C1, C2) :-
   exists has_not_both1(P, C1, C2, C3).
has_not_both1(P, C1, C2, C3) :-
   has_not_all_three(P, C1, C2, C3),
   has(P, C1).
has_not_both1(P, C1, C2, C3) :- .....
has_not_both1(P, C1, C2, C3) :- .....
has_at_least_one(P, C1, C2) :-
   exists has_at_least_one1(P, C1, C2, C3).
has_at_least_one1(P, C1, C2, C3) :-
   has_at_least_one_of_three(P, C1, C2, C3),
   has_not(P, C1).
has_at_least_one1(P, C1, C2, C3) :- .....
has_at_least_one1(P, C1, C2, C3) :- .....
has(P, C) :- exists owns(P, C, C1).
owns(P, C, C1) :-
   has_at_least_one(P, C, C1),
   has_not(P, C1).
owns(P, C, C1) :-
   has_at_least_one(P, C1, C),
   has_not(P, C1).
has_not(P, C) :- exists owns_not(P, C, C1).}
```

We omit for the sake of brevity the definitions of other relations, which indeed are straightforward. Notice the presence of deduction rules for the predicates `has` and `has_not`. These rules are needed since there are several chances to state that a player `P` has got (resp. has not got) a certain card `C`, beyond the fact that the knowledge base contains the assertion `has(P,C)` (resp. `has_not(P, C)`).

Up to this point we have defined the logical part of the whole system, that is the theory `choice` containing the deduction rules allowing either to find the solution or to establish new requests, and the theory `infer` allowing to deduce new facts from the knowledge base. Of course, this logical part should contain a theory, say `InitialKB`, containing a fact `has(player(1),C)` for each card `C` `player(1)` is given at the very beginning.

We are now in the position to define the functional part of the system, which implements the actual game playing. First of all, let us assume that the interaction between `player(1)` (i.e. our program) and the other players takes place by means of an input stream, defined as a list which can be consumed by the program itself (this approach to the modelling of input/output by means of streams is typical of non strict functional languages (Richards 1982)). In our case, after a request by `player(1)` of the kind `(C1, C2, C3)` the first element of the input stream will be `has(j,C)` with `C=C1` or `C=C2` or `C=C3`, corrsponding to the fact that `player(j)` possesses card `C`. Analogously, after a request by `player(n)`, `n≠1`, of the kind `(C1, C2, C3)` the first element of the input stream will be `has(player(j),-)`, corresponding to the fact that each player, including `player(1)`, knows who gave the answer, but only the asking person knows which particular card `player(j)` possesses. Moreover, let us assume that the number of players is defined by the constant `N`.

The functional part contains a function FromMyGuess which, given a triple of cards (P, L, W) denoting the current question raised by player(1), yields a logic theory containing the new facts which player(1) is able to derive from the answer to the question. The core of FromMyGuess is the auxiliary function FromAnswer which deduces a collection of facts involving the predicate has_not from the fact that no player before a given one answered the question (we assume the ordering player(1) < player(2) <...< player(N)). Notice the definition by cases of this recursive function.

```
fun FromMyGuess P L W Input =
   let   has(i, C) :: Rest = Input;
   fun   FromAnswer 1 = EmptyTheory |
         FromAnswer k =
         union({has_not(player(k), susp(P));
                has_not(player(k), loc(l));
                has_not(player(k), weap(W))},
               FromAnswer (k-1));
   in
    (union( (has(player(i), C)),
            FromAnswer (i-1)),
     Rest)
```

The function FromOthersGuess is analogous to FromMyGuess, except that the derived facts are of different kind since player(1) is able to know only part of the answer given to the request raised by other players. The definition is omitted for the sake of brevity. Notice that FromMyGuess (as well as FromOthersGuess) returns a pair (T, R) where T is the theory containing the new facts inferred and R is the rest of the input stream.

What follows is the definition of the main function of the program, implementing player(1)'s turn, which returns a value of the following type result

```
datatype result = win of int | loss of int;
fun Turn KB Input =
   let
      val KB1 =
      union(choice, union(KB, infer));
      fun IsEmpty Set = (Set = {});
      val Answer =
            {(P, L, W) |
              is_in_the_box(susp(P)),
              is_in_the_box(loc(L)),
              is_in_the_box(weap(W))
             wrt KB1};
   in
   if not IsEmpty Answer  then  win(1)
   else  let
      val (P,L,W)=
            choose({(C1,C2,C3) |
             request(C1,C2,C3) wrt KB1};
      val (NewFacts, Rest) =
            FromMyGuess P L W Input;
      val NewKB = union (KB1, NewFacts)
   in OthersTurn NewKB Rest
```

The function OthersTurn takes care of increasing the knowledge of player(1) when it is the turn of the other players. It can be defined in a very similar way as function Turn, making use of function FromOthersGuess. Finally, the function choose randomly selects an element of a set (alternatively, some suitable heuristics can be adopted for choosing an item).

The example points out how the logical and functional components of LML play the appropriate role in making the construction of Knowledge Based Systems more expressive. Indeed, the playing strategies have been coded, in a natural way, using logic theories whereas the actual process of playing, which is intrinsically algorithmic, has been coded using the functional component. The example sheds also light on how LML is apt to handling situations in which knowledge is split into modules and evolves over time.

4 IMPLEMENTATION TECHNIQUES

The architecture chosen for our interpreter (Fig. 1) is rather traditional. First, input from the user passes through a parser, which checks its syntactic correctness and builds up a representation of it in an internal form. The transformed input is then passed to the type checker, whose role is to check the static semantic correctness. Finally, the phrase is passed to the executor, which evaluates it, using two sub-modules. The logical one implements the intensional operators on theories and SLDN resolution.

The Environment records all information about the objects currently known to the system. It is used by all the modules of the system. Note that even the parser use the environment, since in situations such as formal parameters analysis and type forcing we have to know whether a syntactic item is a data constructor or not.

The system has been implemented in NIP Prolog on a SUN workstation. This means that LML is not a winner in performance. Nevertheless, as we have experimented, a careful implementation of SLDN, together with delayed evaluation of negative information (see later), improves efficiency considerably.

Let us now see some specific aspects of the prototype. We will not give a deep description of the details of the interpreter. Instead, we shall see some aspects peculiar to the implementation (and some of the problems we have encountered), to show the design philosophy underlying LML.

4.1. Type checking

LML highly resembles ML in its syntax and in the

interaction with the user. If one is using only the functional constructs of LML, he would not be able to distinguish it from ML, unless for some minor restrictions on the syntax, which do not affect the expressiveness of the language. Hence, the type checker is basically that of ML (Cardelli 1985).

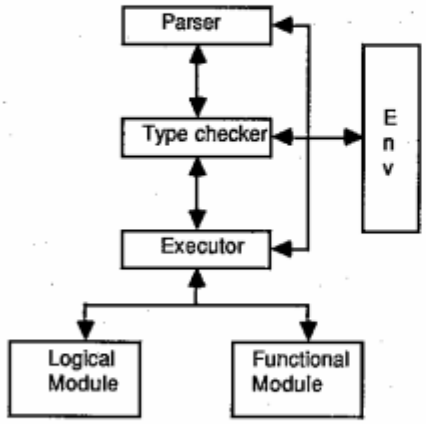

Fig. 1

Type checking a theory is rather different from the same operation on, say, a **letrec** (the functional construct semantically closer to a theory definition) because when one wants to determine the signature of a theory, a fixed-point operation on the types of the predicates has to be carried on. For example, in the following fragment of a theory:

```
p(x) :- q(x).
q(z) :-
```

one has to say that the type of the argument of p must not be more general than that of q, but this imposes no constraints on the type of q, which may be any type. This is different from the typing of functions in ML. In the fragment:

```
letrec
    fun f x  = x + (g 1) and
    fun g x = 1
```

g would come to have the type **int -> int**, even if its definition does not make any restriction on the domain. This depends on the implementation of the type checker: the type of g (and f) in a **letrec** is determined considering all usages of g in the letrec. Notice that in ML, because of its block-structure, a fixed-point operator is not required, since the declaration:

```
let fun g x = 1 in let fun f x = x + (g 1)
in . . .
```

could be used to obtain the desired (polymorphic) type.

The type checker performs this fixed point operation (on theories), then maintains this information, since the application of an intensional operator to two theories as well as a set expression may yield a violation to such type constraints.

4.2. Handling laziness

Another aspect of the interpreter which distinguishes it from Standard ML is *laziness*. Laziness is a computation rule that says "Don't evaluate anything unless you can't proceed without the result of the evaluation". With laziness one can handle infinite structures without necessarily diverging.

LML tries to push laziness to its extreme. Basically one can say that LML does not evaluate anything, unless explicitely requested from the user. When the *printer* asks for some part of a value to print, it starts the evaluation process, which evaluates just one value. The process propagates backwards, asking for just the evaluation of small pieces of the composed values, until the value has been instantiated enough to satisfy the request.

One can use laziness for different purposes. The implementation of SLDN is one of these. If one tries to implement it as described in the preceding pages, the user will be submerged by a lot of clauses (of the order of a hundred to a thousand for a program the size of cluedo) produced by the mere application of the negation rule. This is a time and space consuming operation, which in some cases is even unnecessary (negative information is not always needed for all predicates).

What we have done is to delay the evaluation of the negative information for a predicate, doing it only when we are faced with the request to solve a negative goal involving that predicate. The negation of that predicate (and only of that) is then evaluated, and we can now try to satisfy the negative goal.

Another place where we are lazy is in presence of *nafs*, which are delayed until the end of the goal, in order to have their variables instantiated as much as possible. The reason is that a *naf* is evaluated exploiting the **not** of the underlying Prolog, which explores the whole refutation tree for the predicate. More instantiated is the predicate, less solutions are possible. Delaying the *nafs* reduces, in many cases, the solution space to be searched.

5 CONCLUSIONS

This paper has presented LML and its use from a pragmatical viewpoint. On the other hand much effort has been spent to provide the language with sound theoretical foundations. Two are the critical issues: proving that the intensional negation operator, along with the SLDN resolution, are a correct way of handling negative information and finding a suitable denotational domain for LML theories, amenable of being the basis for a complete denotational semantics of the language.

As to the first problem, the results in (Barbuti et al. 1987, Barbuti et al. 1988, Mancarella 1988) give the correct

solution, allowing to state the soundness and completeness of SLDN with respect to programs completions, in the case of definite Horn theories, while work is still in progress for the general case of theories using negation and universal quantification within clause bodies. On the other hand, the domain for LML theories is currently under investigation. The most promising idea is to consider the space of continuous mappings from Herbrand interpretations to Herbrand interpretations as the semantic domain. Each mapping is actually a pair $<T^+, T^->$ of mappings, which, roughly speaking, mirrors the idea that a logic theory has both a positive and a negative component (the latter being the intensional negation of the former). Hence, the denotation of a logic theory P is the pair $<Tp^+, Tp^->$ where Tp^+ is the *immediate consequence operator*, as defined by Apt and vanEmden (1982), associated to P, whereas Tp^- is the one associated to its intensional negation (of course the definition of Tp has been suitably extended to general programs). Some preliminary results on this issue can be found in (Pedreschi 1988, Mancarella and Pedreschi 1988).

The definition of the language itself and its execution environment is still in progress. Among several interesting issues one is worth to be mentioned, i.e. finding a neat and coherent way of embodying in the functional layer primitives for the metainterpretation of logic theories, like, for example, *clause* and *call* in Prolog.

REFERENCES

Apt,K.R. and vanEmden,M.H. "Contributions to the Theory of Logic Programming". *Journal of the ACM*, 29,3 (1982) 841-862.

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

Barbuti, R., Mancarella, P., Pedreschi, D. and Turini, F. "A Transformational Approach to Negation in Logic Programming", to appear in *Journal of Logic Programming* (1988).

Bowen, K.A. and Kowalski, R.A. "Amalgamating Language and Metalanguage in Logic Programming", in *Logic Programming*, Academic Press (1982), 153-172.

Cardelli, L. "Basic Polymorphic Typechecking", in: Polymorphism, vol. II,1 (1985).

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

Darlington,J.,Field,A.J. and Pull, H. "The unification of Functional and Logic languages", in *Logic Programming: Functions, Relations and Equations*, Prentice-Hall (1985).

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

Henderson, P. *Functional Programmin: Application and Implementation*, Prentice-Hall (1980).

Kowalski, R.A. *Logic for Problem Solving* (Elsevier North Holland, New York, 1979).

Kowalski, R.A. "Logic Programming", in: Proc. IFIP'83 (North Holland,1983) 133–145.

Lassez, J.-L. and Marriot, K. "Explicit and Implicit Representation of Terms Defined by Counter Examples", *Journal of Automated Reasoning* 3 (1987).

Lloyd, J.W., *Foundations of Logic Programming* (Springer Symbolic Computation Series, Berlin, 1984).

Lloyd, J.W. and Topor, R.W. "A Basis for Deductive Data Base Systems", *Journal of Logic Programming*, Vol. 2,2 (1985) 93–103.

Lloyd, J.W. and Topor, R.W. "A Basis for Deductive Data Base Systems II", *Journal of Logic Programming*, Vol. 1 (1986) 55–67.

Mancarella, P. *Intensional Negation of Logic Programs*. Ph.D. Thesis, University of Pisa (in Italian) (1988).

Mancarella,P. and D. Pedreschi. "An algebra of Logic Programs". in Proc. of Fifth International Conference, Symposium of Logic Programming, Seattle (1988) 1006-1023.

Mancarella,P., Martini,S. and Pedreschi,D. "Complete Logic Programs with Domain Closure Axiom". To appear in *Journal of Logic Programming* (1988a).

Mancarella,P., Pedreschi,D. and Turini,F. "Functional Metalevel for Logic Programming", in: D.Nardi and P.Maes (eds.), *Meta-Level Architectures and Reflections*, (North-Holland, Amsterdam, 1988b) 329–344.

Milner, R. "A proposal for Standard ML", in: Proc. of 1984 ACM Symp. on LISP and Functional Programming (1985) 184–197.

Pedreschi, D. *Logic Programming: Compositional Semantics, Algebraic Structures and Complete Programs*. Ph.D.Thesis, University of Pisa (in Italian) (1988).

Reiter, R. "On closed world data bases", in: H.Gallaire and J.Minker (eds.), *Logic and Data Bases* (Plenum Press, New York, 1978) 55–76.

Richards, H. "The pragmatics of SASL for programming applications", Technical Report ARC 82-15, Austin Research Center, Borroughs Corporation (1982).

Robinson, J.A. and Sibert,E.E. "LOGLISP: an alternative to PROLOG", in *Machine Intelligence 10*, (1982).

Sato,T. and Tamaki,H. "Transformational Logic Program Synthesis", Proc. Conf. on Vth Generation Computer Systems (1984).

Shepherdson, J.C. "Negation as Failure: a Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption", *Journal of Logic Programming*, Vol. 1,1 (1985)