Estratto da

R. Ferro e A. Zanardo (a cura di), *Atti degli incontri di logica matematica* · Volume 3, Siena 8-11 gennaio 1985, Padova 24-27 ottobre 1985, Siena 2-5 aprile 1986.

Disponibile in rete su http://www.ailalogica.it

FUNCTIONAL META LEVEL FOR LOGIC PROGRAMMING
(extended abstract)

DINO PEDRESCHI, PAOLO MANCARELLA, FRANCO TURINI
Dipartimento di Informatica Università di Pisa

1. Introduction

Logic programming is, nowadays, one of the most widespread paradigm for knowledge representation [4, 2]. The reasons are to be found in its theoretically well established background and the better and better implementations which are launched every day on the market.

If knowledge engineering has to supersed, in the future, standard programming as a better means of exploiting computers, knowledge engineering environments have to allow the construction and the manipulation of large knowledge bases in an orderly way. Put another way, if the tools and the methodologies for *programming in the large* have been a (partial) solution to the construction of large conventional software systems, there is greater and greater need for tools and methodologies for *knowledge engineering in the large*.

Our work aims at this goal, providing an environment where chunks of knowledge can be represented as logic programming theories and the theories can be manipulated via operators embedded in a functional programming language. In other words, logic theories are added as first class objects to a functional, ML-like, programming language. The functional layer can look at the logic theories in two modes:

- **estensionally**. According to this mode, logic theories can be queried in the usual way allowing to evaluate sets of tuples which can be further manipulated by the functional layer. This mode provides an integration between logic and functional programming similar, in spirit, to the one proposed by Robinson for LogLisp [6].

- **intensionally.** According to this mode, logic theories are considered as first class objects. They can be passed around as functional arguments and **intensional** operators can be applied to them. The class of intensional operators include an intensional negation operator, which, given a logic theory, yields a new logic theory which computes the effective complement of the original one. Other operators allow to join theories together, to intersect them etc..
This mode allows to dynamically manipulate chunks of knowledge and, in our opinion, provide a semantically sound way of *knowledge engineering in the large*.

The rest of the paper deals with the presentation of LML, a language defined around the ideas described so far, with hints on its mathematical and operational semantics.

2. LML

The key idea of LML is to allow also logic programs (theories) to be first class objects as well as functions. This is achieved introducing the type of logic programs equipped with a set of operations to create, use and combine theories. The analogy between these operators and the usual operators on functions is summarized in the following picture.

functions	**logic theories**
λ-abstraction	Horn clauses
application	set-expressions
composition	union, intersection, negation

The link between the functional layer and the logic layer is achieved through set-expressions which allow to compute sets by querying logic programs. The set-expression

$$\{x|\ G_{[x]} \text{ w.r.t. } T\}$$

denotes the set of values resulting from the evaluation of the goal G within the theory T.

Since set-expressions can evaluate to infinite sets, a lazy evaluation rule must be taken into account.

The operators on logic theories allow to define a suitable class of intensional operators on sets, thus providing an intensional calculus over denotations of sets which is one of the most actractive features of the language.

In this respect, a very critical issue is finding a suitable treatment of negation in logic programs which, in turn, allows to define in a precise, semantically clear way some of the set-operators.

The approach followed in LML, called **intensional negation**, is based on a transformation technique which allows to explicit the negative information implicitely embedded within a logic program [1].

For each predicate symbol p, intensional negation syntetizes the clausal definition of a new predicate p~ such that p~ holds iff the proof of p(t) finitely fails under SLD resolution.

A new refutation strategy, called SLDIN-resolution, has been provided which is sound and complete with respect to intensional negation.

With intensional negation a complete symmetry in handling both positive and negative knowledge is achieved, allowing to express both positive and negative queries as

existentially quantified formulas. Notice that the notion of *allowed queries*, critical when *negation as failure* [3] is adopted, makes no sense anymore in this context.

The following is an example of intensional negation which can provide the flavor of the transformation technique, which is inductively based on the ability of negating basic terms, i.e. composition of constructors.

Consider the following definition of the predicate LessOrEqual

LessOrEqual (0,X) ←

LessOrEqual (s(X),s(Y)) ← *LessOrEqual* (X,Y)

Intensional negation yields the following clauses

LessOrEqual ~(s(X),0) ←

LessOrEqual ~(s(X),s(Y)) ← *LessOrEqual* ~(X,Y)

Intuitively, intensional negation builds a program which succeeds when the positive program finitely fails. Finite failure is caused, at the end, by the fact that no clause exists which is unifiable with the current goal. The idea is then to find out for the negated predicate which terms do not unify with the terminal clauses. In the example, the term s(X) can never unify with 0.

Due to intensional negation, each logic theory is implicitly extended with the clauses defining the effective complement of each predicate symbol. Whenever two logic theories are combined together, the negative information is taken into account. As an example, the union of theories T_1 and T_2 is computed by merging the clauses for the original predicates and, then, syntetizing with intensional negation the negative knowledge. Intersection, in a dual way, is computed by merging the negative knowledge and, then, syntetising with intensional negation the new positive information.

Polymorphic types and type-checking in the ML-style has been added [5]. In the LML approach the result is obtained very smoothly by carrying over the typing features of the functional layer to the logic layer.

Finally, it is worth mentioning that the mathematical semantics of the logic layer is built on top of a domain which, again, takes into account both positive and negative knowledge coded within logic theories, thus permitting continuous interpretations for the intensional operators. In this context, some common extralogical features of widespread logic programming systems (e.g. the *assert* and *retract* Prolog builtins) are semantically well understood.

3. Conclusions

LML offers a functional meta-level on top of logic programming in two ways:

- exploits the logical layer as a way of extensionally computing sets,
- allows the intensional manipulation of logic theories as a formal way of manipulating chunks of knowledge.

In summary, the motto of LML is *putting logic theories together in a functional style.*

References

[1] R.Barbuti, P.Mancarella, D.Pedreschi, F.Turini, "Intensional Negation of Logic Programs", submitted for publication to *J. of Logic Programming* (1986).

[2] A.Barr, E.A.Feigenbaum (Eds.), *The handbook of Artificial Intelligence*, Vol. 1, Pitman, London (1981).

[3] K.L.Clark, "Negation as Failure", in *Logic and Data Bases*, H.Gallaire and J.Minker (Eds.), 292-322, Plenum Press (1978).

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

[5] R.Milner, "A proposal for Standard ML", Proc. of *1984 ACM Symp. on LISP and Functional Programming*, 184-197 (1984).

[6] J.A.Robinson, E.E.Sibert, "LOGLISP: Motivations, Design and Implementation", in *Logic Programming*, K.L.Clark and S.A.Tarnlund (Eds.), 299-314, Academic Press (1982).