Logic Meta-Programming Facilities in 'LOG

I. Cervesato[*], G.F. Rossi[**]

(*) Dip. di Informatica, Univ. di Torino - C.so Svizzera, 185 - Torino (I)
(**) Dip. di Matematica, Univ. di Bologna - P.zza Porta S.Donato, 5 - Bologna (I)

Abstract. A meta-level extension of a logic programming language is presented. The resulting language, called 'LOG (read *quote-log*), provides meta-programming facilities similar to those of Prolog while preserving a declarative logical semantics. It also offers new meta-programming opportunities as compared with Prolog due to its ability to treat whole programs, i.e. sequences of clauses, as data objects. The extension basically consists in defining a suitable *naming scheme*. It associates *two* different but related meta-representations with every syntactic object of the language, from characters to programs. The choice of the double meta-representation is motivated by both the user and the implementation viewpoints. All Prolog built-in meta-predicates can be redefined as 'LOG programs by exploiting the new naming scheme. Then some syntactic sugar is added to make the language more concrete. Some examples are given, in particular to show the ability of the language to deal with programs as data.

1 Introduction

The problem of meta-programming in the context of logic programming was systematically faced for the first time by Bowen and Kowalski in [3]. Since that time, a large number of researchers have carried this idea on in many directions. Relatively few efforts, however, have been devoted to the design of an *effective* logic programming language equipped with meta-programming capabilities similar to those usually available in Prolog but defined in a *cleaner* way. Among them, we must mention MetaProlog [2, 4], and, more recently, the Gödel language [6].

This paper moves along these lines and leads to the definition of an extended logic programming language - called 'LOG - which provides meta-programming facilities similar to (or, possibly, better than) those of Prolog. It has the very same aims as *Gödel*, at least as far as the introduction of meta-programming facilities is concerned: "... to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog" [6]. 'LOG is also similar in aims to Barklund's proposal [1]: defining "a naming of Prolog formulas and terms as Prolog terms to create a practical and logically appealing language for reasoning about terms, programs, ...".

Also the applications we have in mind are mostly the same as those of the mentioned proposals, namely the development of software tools (the meta-programs) that manipulate other programs (the object programs) as data, such as debuggers,

Work partially supported by M.U.R.S.T. 40% and C.N.R. - Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", grant no. 890002369.

compilers, program-transformers, etc.. We do not consider as part of our language any reflection mechanism which would allow a meta-representation to be obtained from the object it denotes or vice versa. This differentiates (both in aims and in nature) our proposal from others, such as Reflective Prolog [5] and R-Prolog* [15], that, on the contrary, assume a reflection mechanism to be available, though not visible at the user level.

The main problem is that of defining a suitable *naming scheme* by which the syntactic entities of the language can be referred to and manipulated at the meta-level. Here we stress the fact that naming should apply to *every* syntactic entity of the language, from characters to programs. In addition, we want the naming scheme to be *effective*, that is to burden not too much the user with an heavy notation, and to allow *efficient implementations* of the language to be devised.

The key idea underlying our proposal is to provide *two* different but *related* meta-level *representations* for each syntactic object of the language. Precisely, the meta-level representations consist of a constant *name* and a structured ground term, called the *structural representation*. The name describes an entity as a whole, while the structural representation describes the structure of the entity in terms of the *names* of its components, thus allowing one to explore its internal structure. Moreover, for each composite syntactic object, it is possible to relate its name to its structural representation by means of the predefined predicate <=>. While neither meta-representation is especially original on its own, using them together seems to offer quite interesting possibilities.

The idea of a double meta-representation was already applied in a more pragmatical sense and to a limited extent (programs only) to the definition of the meta-logical facilities of the *EnvProlog* language, an *extended Prolog* aimed at building Prolog programming environments [10, 11, 12]. In this paper, we start instead with a pure logic programming language and we apply the naming scheme to every syntactic entity of the language. Then we show that a more concrete version of the language embodying this naming scheme can be obtained by the addition of a suitable syntactic level; this makes the language easier to be handled both for the user and the implementation. We show also that the resulting language exhibits higher meta-programming attitudes than Prolog (in particular, as regards the ability to deal with programs as first-class data objects) while preserving a logical reading.

Section 2 presents the main features of the naming scheme provided by 'LOG: names, structural representations and the <=> operator used to relate the name and the structural representation of each syntactic object. The usage and motivations of the double meta-representation from the user viewpoint are discussed in Section 3. Section 4 discusses how usual Prolog built-in meta-predicates can be redefined in 'LOG. Section 5 presents the syntactic additions and conventions we assume for the concrete version of 'LOG. The ability of our language to deal with programs as data is highlighted in Section 6 by showing some simple examples. Finally, Section 7 briefly discusses the implementation issue, pointing out some motivations for the use of the double meta-representation also from the language implementation viewpoint.

2 Meta-Representations

'LOG syntax is mostly the usual syntax of logic programming languages (cf. for instance [9]) and will be skipped here, except for those parts concerning the meta-representations.

We start with an ordinary Horn clause language and we conservatively extend it to one in which every syntactic entity is named by ground terms of the language. Precisely, each 'LOG syntactic object has *two meta-representations* associated with it, called the *name* and the *structural representation* of the object.

2.1 Names

The *name* of an object is a *constant* symbol which is isomorphic in structure to the object it refers to. If e is a syntactic expression of the language, 'e' is its name. For example,

```
'append([],X,X).
 append([A|X],Y,[A|Z]) :- append(X,Y,Z)'
```

is the name of a program defining the usual append predicate which concatenates two lists. As another example, 'f(a,g(X))' is the name of the term f(a,g(X)).

Objects having a name in 'LOG are programs, clauses (including goals), terms, symbols and characters. Accordingly, names are partitioned into five different classes: program names, clause names, term names, symbol names and character names. Notice that these classes are not necessarily disjoint. The same name, in fact, can denote different syntactic expressions depending on the context where it is used; for instance, 'alpha' can represent either a symbol or a term with no arguments or a clause with no body or a single clause program. Also notice that we do not consider atomic formulas as a syntactic class of the language. Indeed it seems more appropriate to the meta-programming paradigm we are considering here to treat atoms simply as terms.

2.2 Structural Representations

Every composite syntactic object (i.e., symbols, terms, clauses and programs) has a second meta-representation associated with it, called the *structural representation*. This meta-representation is a ground term which describes the structure of the object it denotes in terms of the *names* of its components.

If $e = e_1 e_2 \ldots e_n$ is a syntactic expression where $e_1, e_2, \ldots, e_n$ are its component sub-expressions then the structural representation of e is $['e_1','e_2',\ldots,'e_n']$.

For instance, if $P = C_1.C_2. \ldots .C_n$ is a program then $['C_1','C_2',\ldots,'C_n']$ is the *program structure* of P where $'C_i'$ is the clause name of the clause C_i. Similarly, if f(a,g(X,b)) is a term, the corresponding *term structure* is ['f','a','g(X,b)']. The only exception is the structural representation of clauses. *Clause* structures (other than goal clause structures) rely on the reserved symbol clause for distinguishing the head from the body part (e.g. clause('p',['q','r']) for p:- q,r). In Section 5 we will introduce a *synthetic notation* for structural representations which is more convenient for the user (in contrast with the list notation, also called explicit notation, presented here). Since it is simply syntactic sugar it can be ignored for the moment.

While names are constant symbols (hence atomic entities), structural representations are compound ground terms. Therefore, one can easily define terms similar to structural representations apart from the occurrence of *meta-level variables* in place of some of the names 'e_i' of its component sub-expressions. Such terms will be considered as *partially specified* structural representations. For instance, ['f',X] is not a term structure; however, if the meta-level variable X is instantiated to the name of some term we get a complete term structure, e.g. ['f','a'], ['f','g(b)'], and so on.

Meta-variables in an incomplete structural representation are dealt with as real variables in contrast with object level variables that are frozen inside the names that constitute the structural representation. Therefore the two term names 'f(X)' and 'f(a)' cannot unify at the meta-level, whereas ['f',X] and ['f','a'] unify, yielding the substitution X = 'a'.

Names and structural representations are syntactic entities; therefore they have a name and a structural representation too. For instance, the name of the term name 'Alpha' is ''Alpha''. Thus, 'LOG supports the definition of an infinite tower of meta-levels. Anyway, meta-levels are strictly separated: at each level, the syntactic entities of the lower levels are visible through their names only; variables do not make an exception to this rule. No reflection mechanism is supported by 'LOG.

2.3 Relating Names and Structural Representations

The name and the structural representation of an object can be related to each other by the use of the predefined predicate <=> (written infix), called the *destructuring* or simply the *double arrow* operator.

The *informal semantics* of <=> is: a goal N <=> S is true if N is the name of an object o and S is the ground structural representation of the same object o. Thus, <=> simply defines a binary relation, called the *destructuring* relation, between names and structural representations, i.e. between syntactic expressions of the language.

Actually, we have distinguished five different classes of name symbols. It follows that we must distinguish among different forms of the *generic operator* <=>, accordingly to the different types of its arguments. We will use the four different operators <=p=>, <=c=>, <=t=> and <=s=> for programs, clauses, terms and symbols respectively, still using the generic double arrow operator when speaking of its properties in general and no ambiguities arise (we will see in Section 5 that these differences can be hidden by an upper syntactic level). Here are two simple examples of goals involving <=>:

```
?- 'p :- q,r.  q.  r.' <=p=> ['p :- q,r','q','r'].
yes.
?- 'f(g(a),b,C)' <=t=> ['f',A,'b','C'].
A = 'g(a)'.
```

The second goal succeeds provided the meta-variable A is instantiated to 'g(a)'.

2.4 Semantics

The main differences in the semantics of 'LOG w.r.t. the standard case (as described for instance in [9]) are due to the presence of the *double arrow* operators.

As regards the *declarative semantics* of 'LOG, first a privileged interpretation domain resulting from suitable modifications to the classical Herbrand universe is defined then the privileged interpretation of <=> is given as a relation over this domain.

The modified *Herbrand universe* H is defined in almost the same way as usual except that it is built out of the set of characters composing names besides the set of function and constant symbols that occur in the program in such a way to include all the names and ground structures which can be constructed in that program.

The *privileged interpretations* of the <=> operators are defined as *binary relations* over such H. In particular, for any TN, TS $\in$ H, whether TN <=t=> TS holds or not can be established by: TN has the form 't', TS has the form ['f','t_1',...,'t_n'], t is a term, and t = f•(•t_1•,•...•,•t_n•) where • is the usual *string concatenation* relation and = is the usual syntactic equality. Similar definitions can be given for <=p=>, <=c=> and <=s=>.

Procedurally, a goal N <=> S succeeds from a program P if either N is a name and there exists a ground instance S' of S such that <N,S'> is in the destructuring relation, or N is a variable, S is ground and there exists an instance N' of N such that <N',S> is in the destructuring relation. If, on the contrary, N is a variable and S is not ground then the goal is unsolvable and its proof is delayed till either one of the above cases occurs.

A *refutation* of P $\cup$ {G} is a finite derivation G, G_1,...,G_n of P $\cup$ {G} such that the last derived goal only contains destructuring goals in unsolvable form and there exists a substitution θ which makes all them true simultaneously. A *computed answer* for a refutation of P $\cup$ {G} is now a pair <σ,C> where σ is a substitution for the variables in G computed as in the standard case and C is the (possibly empty) set of destructuring goals in unsolvable form.

Delaying the solution of a goal containing the <=> operator allows a *declarative* reading of programs to be preserved. The order of literals in a clause or in a goal is immaterial. For example, the goal

```
?-  S  <=s=>  ['a','l'|X],  X  =  ['p','h','a'].
```

succeeds with computed answer substitution S = 'alpha' and no destructuring goal left unsolved. If, on the contrary, at the end of the computation a goal of the form N <=> S cannot be solved because N is a variable and S is not ground then N <=> S is returned as part of the computed answer: it will be considered as a *constraint* on values the not yet instantiated meta-variables occurring in it can assume. For example, in

```
?-  N  <=t=>  ['f'|A],  A  =  ['a'|B].
A  =  ['a'|B],
N  <=t=>  ['f','a'|B].
```

there are obvious valid instances of the meta-variables B and N, but not all of them are viable. Actually the way the double arrow operators are dealt with in our proposal can be viewed as a simple form of Constraint Logic Programming [7, 8].

3 Using the Meta-Representations

Names and structural representations are two descriptions of a syntactic object at two different levels of abstraction. There are circumstances in which the inner structure of the object we want to refer to is not important at all. For instance, when writing a procedure for appending two programs we just need to know that we have to append lists of clauses, without getting into their internal details. In other cases, on the contrary, it is important to access the inner components of the syntactic object we have to deal with.

The two different meta-representations 'LOG supplies are intended to satisfy these two different uses. The structural representation of an entity describes the structure of the entity in terms of the names of its components, which, on the contrary, are viewed as monolithic entities. Thus, for instance, the clause

```
p(X,f(a)) :- q(X),r(a,b)
```

can be represented in 'LOG as

```
clause('p(X,f(a))',['q(X)','r(a,b)'])
```

while in other proposals using structural descriptive names only (e.g. [1, 5]) also sub-components are represented as structured terms. For example, according to Barklund's proposal [1], the above clause should be represented as

```
clause(atom(p,[var(0),compound(f,[const(a)])]),
       conj(atom(q,[var(0)]),atom(r,[const(a),const(b)])))
```

Notice that if symbols had a name too then they should be replaced by the structured terms, e.g. lists of characters, representing them.

Having the structural representation only, it may result quite cumbersome for the user to represent such entities as programs and clauses at the meta-level, and, on the other hand, it may result quite expensive for the implementation to maintain the structural representation of low level entities, such as symbols. Actually, proposals which use a structural meta-representation only usually do not cover the naming of all the syntactic entities of the language: they usually exclude the two extremes, namely programs and symbols. The use of a synthetic notation as a shorthand for complex structured names, such as the one proposed in [5], solves only the problem of notational conciseness but still leaves the implementation problems unsolved (the implementation issue will be briefly addressed in Section 7).

Whenever a deeper detail level is needed, 'LOG provides the user with the <=> operator. Given the name of an entity, one can obtain its structure by applying the proper <=> operator to the name. Thus, for instance, if we want to know which is the name of the predicate defined by the above clause we can go inside the clause structure by unification and then apply <=t=> to its first argument. The goal

```
?- clause('p(X,f(a))',['q(X)','r(a,b)']) = clause(H,_),
   H <=t=> [N|_]
```

will instantiate N to 'p'.

As a more comprehensive example, which makes use of the full power of the double arrow operators, we show the definition of a predicate psort which is able to sort clauses of a program according to the names of their head predicates. The arguments of psort are two program names, namely the object program and its sorted version.

```
psort(Prog,Sorted_Prog)  :-
        Prog <=p=> ProgStruct,
        sort(ProgStruct,Sorted_ProgStruct),
        Sorted_Prog <=p=> Sorted_ProgStruct.
sort(L1,L2)  :-  ...
        %true if list L2 is L1 sorted w.r.t. the order
        %relation defined by the predicate order/2
order(Cl1,Cl2)  :-        %true if Cl1 precedes Cl2
        Cl1 <=c=> clause(Head1|_),
        Head1 <=t=> [PName1|_],
        Cl2 <=c=> clause(Head2|_),
        Head2 <=t=> [PName2|_],
        string_comp(PName1,PName2).
```

where the predicate string_comp(PName1,PName2) tests if the atomic symbol S1 precedes S2 w.r.t. a standard order of characters as defined by the list ['a','b',...,'z']. It employs the <=s=> operator to obtain the lists of the characters composing the given symbols and then compares these lists.

It is important to realize that this program does not use any extra-logical feature. To obtain something similar in C_Prolog one should use such extra-logical built-in predicates as clause, =.. and @< (the latter used in place of our string_comp).

4 "Reconstructing" Prolog Built-in Meta-Predicates

All the meta-predicates Prolog usually supplies in the form of built-in predicates are definable in 'LOG, at least in principle, within the language itself. In particular, the object level provability relation of a goal from a program can be defined, even if quite inefficiently, as a 'LOG program, similarly to the definition of the *demo* predicate given in [3]. In this way it is possible, on the one hand, to give these predicates a logic semantics, and, on the other hand, to ignore them while performing a formal analysis of the language.

Actually, some of Prolog built-in meta-predicates, such as =.., name and ==, become unnecessary in 'LOG, since explicit representations of terms and symbols are directly available. For instance, the Prolog clause

```
p(X) :- X =.. [F,a1|Args],q(F).
```

can be replaced in 'LOG by the clause

```
p(X) :- X <=t=> [F,'a1'|Args],q(F).
```

where the argument of p is assumed to be a term name. Now, assume q is defined as q(f) (resp., q('f') in 'LOG). In Prolog the goal p(f(a1)) succeeds while, unfortunately, the goal p(X) fails. In 'LOG, on the contrary, also the goal p(X) succeeds yielding the constraint X <=t=> ['f','a1'|Y]. This establishes that X is constrained to be the name of a term of the form f(a,...). In particular, the solution X = 'f(a1)' can be obtained from this constraint by instantiating Y to [].

Some other simple Prolog meta-predicates, such as var, atom, etc., can be defined quite easily in 'LOG. var, in particular, is concerned with a crucial point, that is*variable naming*. Let us briefly comment upon this point. The name of an object level variable Alpha is 'Alpha'. Its structural representation instead is ['Alpha'] that is a list of a single element which is a symbol name. Given the symbol name, its structural representation can be easily obtained via the <=s=> predicate. Thus we can inspect the internal structure of the symbol and check whether it is a variable or not (we assume the syntactic conventions of most Prolog systems where variables are symbols with initial capital). Therefore, the Prolog meta-predicate var can be redefined in 'LOG as follows:

```
var(TN)  :-  TN <=t=> [SN],
             SN <=s=> [CN|_],
             upperAlphabet(U),  member(CN,U).
upperAlphabet(['_','A','B',...,'Z']).
```

where TN is intended to be instantiated to a term name. The goal ?-var('Alpha') clearly succeeds with this definition. The goal ?-var('f(a)'), on the contrary, fails since the call to <=t=> in var fails. Notice that if TN is not instantiated yet when var is called then the solution of the destructuring goals in var is simply postponed. Thus, the goal ?-var(X) succeeds with computed answer X<=t=>[SN],SN<=s=>['_'|_]; and then, through backtracking, with computed answer X<=t=>[SN],SN<=s=>['A'|_], and then X<=t=>[SN],SN<=s=>['B'|_], and so on. This result establishes that a variable is a non-structured term whose first character is any of '_', 'A', 'B',... . The goal ?-var(X),X='f(a)' clearly fails, whereas the same goal erroneously succeeds in conventional Prolog.

Using 'LOG meta-programming facilities it is also possible to define the *unification* procedure between two object level terms and then use it to define other typical Prolog meta-predicates. The unification procedure can be implemented as a predicate unify(T1,T2,Subs) where T1 and T2 are the names of the terms to be unified and Subs encodes the computed object level variable substitutions as a list of pairs X/t where X is the name of a variable occurring in T1 or T2 and t is a term name. Thus, for instance, the two term names 'f(X,b)' and 'f(a,Y)' do not unify at the meta-level but they unify at the object level: unify('f(X,b)','f(a,Y)',S) succeeds with S = ['X'/'a','Y'/'b'].

Using unify it is easy to define, for instance, extended versions of the call and clause built-in predicates of ordinary Prolog, where the program to work with and the generated substitutions are handled explicitly as new arguments of the predicates. Following EnvProlog [10], we call these meta-predicates ecall and eclause, respectively. In particular, the ecall predicate is defined as follows:

```
ecall(PN,GN,Subs,C)
```

holds if the goal represented by GN can be proved in the object level program represented by the program name PN. Subs is a (possibly empty) list of pairs X/t representing the substitutions of the object level variables occurring in G and C is a (possibly empty) list of constraints, that is destructuring goals N<=>S in unsolvable form, generated by the proof of G. For example:

```
?-  ecall('p(a,Y) :- q(Y). q(f(b))', ':- p(X,Y)',S,C).
S  =  ['X'/'a', 'Y'/'f(b)'],  C  =  [].
```

5 Towards a Concrete Language

Some syntactic sugar can be added to the language described so far to make its implementation more efficient and simplify its use.

First, we extend the syntax of names so to be able to determine for each name which kind of name it is, i.e. which syntactic class the named objects belong to, by simply looking at the name itself. This ability can be advantageously exploited by the language implementation to select *at compile time* the most adequate *internal representation* for each different kind of name (see Section 7 for a discussion of the implementation issue). Furthermore, the appropriate instance of the generic operator <=> can now be selected automatically, accordingly to the kind of its arguments; so the user must be concerned with only a single overloaded <=> operator, letting the language implementation have the task of disambiguating it.

The syntactic conventions we will use for names are summarized in Figure 1. Assuming these conventions, each name uniquely identifies an object of a precise syntactic class. For instance, '{alpha}', '.alpha.', 'alpha' and '/alpha/' represent a program, a clause, a term and a symbol, respectively. Also ground *structural meta-representations* of different kinds can easily be distinguished each others accordingly to the different kinds of their components. For instance, a list of clause names is necessarily a program structure, a list of term names is necessarily a goal clause and so on. As an example, the three similar structures ['b','e','t','a'], ['/b/','e','t','a'] and [%b,%e,%t,%a] can be easily mapped on to the goal clause :- b,e,t,a, the term b(e,t,a) and the symbol beta, respectively.

	objects	names
program:	c1.c2. ... .cn	'{c1.c2. ... .cn}'
clause:	h:-b1, ... ,bn	'.h:-b1, ... ,bn.'
term:	f(t1, ... ,tn)	'f(t1, ... ,tn)'
symbol:	abc	'/abc/'
character:	c	%c

Figure 1

A further step towards a more concrete programming language is introducing a *synthetic notation* for structural representations which is more convenient for the programmer than the list-like explicit notation used so far. The synthetic notation for the structural representation of an expression e closely resembles the corresponding name of e, except that double quotes are used instead of single quotes. For instance, the synthetic structural representation of the program c1.c2. ... .cn is "{c1.c2. ... .cn}", whereas the synthetic structural representation of the term f(t1,...,tn) is "f(t1,...,tn)".

Meta-variables can be easily handled by the explicit notation of structural representations. However, it would be desirable to use meta-variables in the synthetic notation as well. In order to allow object level variables to be easily distinguished from meta-variables also when using the synthetic notation we admit the former to be enclosed in quotes whenever ambiguities might arise. Thus, for instance, the term structures ['/f/',X,'Y'] and [F,'g(X)','X'] can be represented unambiguously in synthetic notation as "f(X,'Y')" and "F(g(X),'X')", respectively.

Moreover, the usual Prolog notation used to represent the rest of a list is easily extended to structural representations in synthetic form. For instance, the (incomplete) term structure ['/f/','a'|R] can be rewritten in synthetic notation as "f(a|R)". As another example, a predicate that concatenates two program structures can be defined as follows (cf. [12]):

```
appendPS("{}",P,P).
appendPS("{C|P1}",P2,"{C|P3}") :- appendPS(P1,P2,P3).
```

Finally, a synthetic notation is introduced also to represent *nested term structures* in a more convenient way. Nested term structures are lists of term structures (rather than of term names) which can be constructed by repeated applications of <=t=>. For instance, given the term name 'f(a,g(b))' the corresponding nested term structure is ['/f/',['/a/'],['/g/',['/b/']]] or, in synthetic form, "*f(a,g(b))". Meta-variables in partially specified nested term structures may occur at any depth in the term. For instance, in ['/g/',['/h/',X]], i.e. "*g(h(X))", X is clearly a meta-variable. Notice that nested term structures where all object level variables are replaced by meta-level variables closely correspond to *non-ground representations* in Gödel [7]. Also notice that, as a special case, "f(X_1,...,X_n)" and "*f(X_1,...,X_n)", $n \geq 0$ and X_1,...,X_n (meta-)variables, are equivalent synthetic notations for the partially specified term structure ['/f/',X_1,...,X_n].

Nested term structures can be advantageously exploited to give an alternative definition of the ecall meta-predicate which provides some form of communication from the object level to the meta-level which turn out to be very useful in practice.

```
ecall(PN,NGS)
```

holds if there is an instance NGS' of the list of partially specified nested term structures NGS such that the conjunction of goals represented by NGS' can be derived, at the object level, from the program represented by the program name PN. For example

```
?-  ecall('{p(X) :- q(X). q(a)}', ["p(X)","q(X)"]).
X = "a".
```

Similarly, it would be possible to define also a version of eclause working with lists of partially specified nested term structures instead of lists of term names and then use it to define the vanilla meta-interpreter in almost the same way as in ordinary Prolog.

6 Programs as Data

One of the most peculiar feature of our language is the ability to deal with whole programs, i.e. finite sequences of clauses, as data objects. In this section, we briefly point out two classes of problems for which program names may result particularly useful. A wider discussion about this topic can be found in [12].

6.1 Program Structuring

A program can contain an assertion a(PN) about another program designated by the program name PN. Therefore, the set of clauses in a program can be partitioned into separate smaller subsets defined as *inner programs*. Inner programs can be dealt with as data by the enclosing program, i.e. by the program at the meta-level. For example, the program

```
alpha  prog '{p(X) :- q(X),r. q(a). r}'.
beta  prog  '{p(b). q(a)}'.
p(X)  :-  q(X).
q(c).
```

where prog is a user-defined infix operator, contains the definition of two inner programs. The three definitions of the predicate p occurring in the three different programs are dealt with as definitions of three distinct predicates, i.e., predicate names in a program are *local* to that program. Predicates in an inner program can be accessed only using meta-predicates such as ecall and eclause. For example, adding the clause

```
demo(N,G)  :-  N  prog  P,ecall(P,G).
```

to the above program, we can issue the goal

```
?-  demo(alpha,"p(X)").
X  =  "a".
```

where simple mnemonic names, such as alpha and beta in the example, can be used instead of program names to refer to programs.

The use of *structural-descriptive* names for representing programs at the meta-level is a major difference with respect to MetaProlog [2, 4]. In MetaProlog theories (i.e., sets of clauses) are named via *simple constant* names with no resemblance of the structure of the object they denote. Thus, "All MetaProlog program databases ... are set up either by reading them in from files or by dynamically constructing them using system predicates" [4]. In 'LOG, on the contrary, a program name lists explicitly clauses that compose it. Thus our solution is well suited to support *program modularization*. Program names can be statically nested at any depth. The global program database can be split into a number of smaller program units, possibly nested, which can be accessed only via meta-predicates such as ecall.

In addition, 'LOG allows meta-variables to occur in program structures whereas the same is not feasible in MetaProlog. As a consequence, it is not possible in MetaProlog to define for instance a predicate like the predicate appendPS of Section 5: such a predicate could be implemented in a rather awkward way as a series of add_to calls. On the other hand, the use of structural descriptive names is not adequate to support the construction of self-referential sentences.

6.2 Clausal Representation of Data Structures

Programs can be used also to collect clauses defining some complex data structure so that it can be managed as any other term (e.g. passed to a procedure as a parameter) maintaining all the advantages of the clausal representation (e.g. access by unification). For example, the following assertion

```
g1 graph '{a(a,b).a(a,c).a(b,c).a(b,d).a(c,d)}'.
```

can be used to define a graph, named g1, where graph is a user-defined infix operator and, as usual, a(X,Y) represents an arc between two nodes X and Y.

With such a representation, it is easy to define general predicates dealing with graphs, such as, for instance, a predicate path(X,Y,G) for finding a path between two nodes X and Y in a given graph G:

```
path(G,X,Y) :- ecall(G,"a(X,Y)").
path(G,X,Y) :- ecall(G,"a(X,Z)"),path(G,Z,Y).
```

Thus, it is possible to solve, for instance, the goal:

```
?- g1 graph G,path(G,a,d).
```

Different graphs can coexist in the same program if each graph is defined as a separate program, possibly with its own mnemonic name. This is much more difficult, and less elegant, to obtain using standard Prolog.

7 Implementation Issues

The naming scheme we have chosen for 'LOG and, in particular, the use of a double meta-representation is justified also by a number of implementation concerns we will try to summarize in this section.

First of all, we notice that the use of meta-level names which are *isomorphic in structure* to the named objects allows the internal representation of meta-level names to be used directly as the internal representation of the objects themselves. No explicit link between objects and object names must be maintained by the system as, on the contrary, it would be necessary if unstructured constant names were used (e.g. in MetaProlog).

Therefore, having two meta-representations for the same object actually amounts to having two different internal representations for the same object. When using the structural representation we are likely to inspect the structural composition of the named object. This requires a *list-like* internal representation to allow standard unification to be used to access components. Conversely, when using the constant name we want to deal with an object as a monolithic entity. No logical operation is allowed to access the internal structure of a constant. A name could be stored in main memory simply as a *string*.

However, an efficient implementation of the language could choose a different internal representation of names without interfering with their structural representation. Indeed, although all Prolog built-in meta-predicates are in principle definable in 'LOG, the concrete version of our language should provide also a low level implementation of most of them in order to obtain acceptable execution efficiency for practical applications. Of course this require an adequate internal representation of the object to be dealt with. In particular, an efficient implementation of meta-predicates dealing with *programs*, such as ecall and eclause, requires program names to be stored in such a way to make accessing clauses as fast as possible. Thus, for instance, program names can be represented internally as a tree-like structure with auxiliary pointers and indexes or hash tables for improving search operations on clauses.

If we had the *structural representation* only, and therefore only the list-like internal representation for every object then there would be an unacceptable decreasing in the overall efficiency of the language. Just think of the overhead (both in space and in time) caused by representing all symbols as character lists. If, on the other hand, we had only *names*, and therefore only special-purpose internal representations, as required by any real implementation to obtain a reasonable execution efficiency, then any attempt to access the structure of a syntactic entity would cause non-trivial difficulties. Ad-hoc operations should be provided in this case instead of standard unification.

With our solution, which internal representation of an object must be used can be established by the user by selecting the appropriate meta-level representation of the object. Given one representation, the other one can be built in a quite straightforward manner. To this regard, notice that a structural representation describes the structure of an object in terms of the *names* of its components. So it is possible to use directly the internal representation of names as the elements of the internal representation of the structured name of an object. Furthermore, one representation can be built from the other one only when it is really necessary, that is, whenever a goal N<=>S is encountered and one of its arguments is a variable while the other one is a ground term. If both the representations of the same object are present a double link is used to connect them to each other.

Finally, notice that the syntactic conventions established in Section 5 allows the language implementation to always select at compile time the appropriate internal representations for each different kind of names and ground structural representations.

8 Conclusions

In this paper we have presented the meta-logic facilities the language 'LOG supplies and briefly discussed their motivations and uses. In particular, we have described a *naming scheme* for logic programs which allows two different but related meta-representations (namely, names and structured representations) to be associated with each syntactic entity of the language, from programs to characters. The availability of such a naming scheme allows all the meta-predicates usually available in Prolog to be defined as 'LOG programs. Also interesting extended versions of some of these predicates (such as ecall, eclause, etc.) can be provided quite easily. Furthermore we have shown simple applications of program names which have no immediate correspondent in standard Prolog.

A simple though inefficient prototype of the language described in this paper has been implemented in C_Prolog. A more concrete and complete version of this language, called *Quote-Prolog*, is under development at present. In particular, the notion of program name, program structure and the <=> operator dealing with them had already been implemented in *EnvProlog* [10, 11, 12] and has been now successfully re-implemented (in a better structured way, and including clause names) in a new extended Prolog interpreter written in Modula 2. It is planned for the near future to continue this implementation to include all the meta-level features of 'LOG.

References

1. J. Barklund: What is a Meta-variable in Prolog?. In: H.D. Abramson and M.H. Rogers (eds.): *Meta-Programming in Logic Programming: Proceedings of the META88 Workshop*, Bristol. MIT Press, 1990, pp. 383-398.
2. K.A. Bowen: Meta-Level Programming and Knowledge Representation. *New Generation Computing* 3, 1985, pp. 359-383.
3. K.A. Bowen, R.A. Kowalski: Amalgamating Language and Metalanguage in Logic Programming. In: K.L. Clark, S.A. Tärnlund (eds.): *Logic Programming*. Academic Press, 1982, pp. 153-172.
4. K.A. Bowen, T. Weinberg: A Meta-Level Extension of Prolog. In: *IEEE Symposium on Logic Programming,* Boston, 1985, pp. 669-675.
5. S. Costantini, G.A. Lanzarone: A Metalogic Programming Language. In: G. Levi, M. Martelli (eds.): *Logic Programming: Proceedings of the 6th International Conference*, Lisbon. MIT Press, 1989, pp. 218-233.
6. P.M. Hill, J.W. Lloyd: The Gödel Report (Preliminary Version). Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991.
7. J. Jaffar, J.L. Lassez: Constraint Logic Programming. In: *Proceedings of the 14th POPL Conference,* Munich. ACM, 1987, pp. 111-118.
8. P. Lim, P.J. Stuckey: Meta-Programming as Constraint Programming. In: S. Debray, M. Hermenegildo (eds.): *Logic Programming: Proceedings of the 1990 North American Conference on Logic Programming*, Jerusalem. MIT Press, 1990, pp. 416-430.
9. J.W. Lloyd: *Foundations of Logic Programming*. Springer Verlag, 2nd ed., 1987.
10. A. Martelli, G.F. Rossi: Enhancing Prolog to Support Prolog Programming Environments. In: H. Ganzinger (ed.): *ESOP'88: 2nd European Symposium on Programming,* Nancy. Lecture Notes in Computer Science 300, Springer Verlag, 1988, pp. 317-327.
11. G.F. Rossi: Meta-programming Facilities in an Extended Prolog. In: I. Plander (ed.): *Artificial Intelligence and Information-Control Systems of Robots-89*, North Holland, 1989
12. G.F. Rossi: Programs as Data in an Extended Prolog. To appear in: *The Computer Journal*, British Computer Society, 1992.
13. S. Safra, E. Shapiro: Meta-interpreters for Real. In H-J. Kugler, (ed.): *Information Processing 86* , North-Holland, 1986, pp. 271-278.
14. L. Sterling, A. Lakhotia: Composing Prolog Meta-interpreters. In: R.A. Kowalski, K.A. Bowen (eds.): *Logic Programming: Proceedings of the 5th International Conference and Symposium*, Seattle, MIT Press, 1988, pp. 386-403.
15. H. Sugano H.: Meta and Reflective Computation in Logic Programming and Its Semantics. In: M. Bruynooghe (ed.): META90: *2nd Workshop on Meta-Programming in Logic Programming,* Leuven, 1990, pp. 19-34.