ENHANCING PROLOG TO SUPPORT PROLOG PROGRAMMING ENVIRONMENTS

A.Martelli and G.F.Rossi

Dipartimento di Informatica - Università di Torino
C.so Svizzera 185 - 10149 Torino (ITALY)
uucp: ...!mcvax!i2unix!leonardo!mrt

Abstract

This paper describes the basic ideas we followed in the development of PROSE, a Prolog programming support environment we are implementing at our Department. We claim that standard Prolog must be adequately enhanced to be well suited to support the construction of an efficient programming environment. For this purpose, some new facilities are supplied by our Prolog (called Env_Prolog) which are mainly intended to allow the language:
- to handle programs as data and to partition the program database into disjoint sets of clauses;
- to support "editing" of clauses in the program database and the controlled execution of Prolog programs. Env_Prolog has been implemented by developing a new interpreter written in the C language. The paper will concentrate mainly on the interpreter and the support it offers to other tools of the environment.

1. Introduction

Programming environment tools for traditional programming languages are very often written in the same language which they are intended to support. This usually requires user programs to be converted into an *internal representation* which can be handled as a data structure of the language itself by all the tools of the environment.
The generation of the internal representation of a program is usually done by a syntax-based editor and has the form of a tree which reproduces the syntactic structure of a program as far as its abstract syntax is concerned.

On the contrary, using a programming language like LISP it is possible to directly view programs as data of the language itself. In this way tools written in LISP can handle user programs directly, without having to resort to any intermediate representation. Advantages of this approach are greater interactivity and expansibility of the environment and simplification of its development process.
Existing LISP environments are usually highly appreciated and demonstrate the effectiveness of this approach.

Prolog shares with LISP many valuable features for the construction of programming environments, such as symbolic manipulation capabilities, dynamic bindings, etc. However standard Prolog does not entirely satisfy the requirement of being able to handle programs as data. Indeed there are several built-in predicates to handle clauses and terms (e.g. *clauses*, *functor*,...) but clauses or programs (= finite set of clauses) cannot be handled as data.

Moreover, standard Prolog [4] does not supply the user with any facility to split a complex program into separate subcomponents ("modules"). Nevertheless this is an important requirement for a programming environment where many distinct programs (i.e. tools and user programs) must cohabit and cooperate. Some Prolog dialects let the language to have such capabilities by extending Prolog with various syntactic structures.

Unlike LISP, only few experiences have been done with the construction of Prolog programming support environments so far. One of the first effort to built such an environment is reported in [5]. Here an internal representation of Prolog is used to face the above problems. It serves also as a way to enrich the program with those informations which are necessary to the environment tools (in particular to the editor and debugger). Unfortunately in this way some of the potential advantages of Prolog with respect to traditional programming languages are missed.

The approach we have taken in the development of PROSE (=Prolog Support Environment) [10,13] is quite different from this one. Our goal has been to allow the whole environment to be written in Prolog without using any internal intermediate representation of programs; at the same time, neither the interpreter should be burdened too much nor Prolog should be extended too much.
The solution we have adopted consists in:

- giving the language the capability to handle programs as data and to partition the program d.b. into disjoint sets of clauses (in the basic form of *theories* as suggested in [2]);
- inserting into the interpreter all the facilities which are required for the development of the environment basic tools and which are very hard or highly inefficient to directly implement in Prolog. This requires the definition of a number of *new built_in* predicates which allow the user to exploit these new facilities.

Selecting which facilities must be supplied by the interpreter and which on the contrary must be implemented within the tools is a critical point. Our criteria has been to reduce as much as possible the number of new facilities that the interpreter must directly support so that its additional burden can be limited and good flexibility is assured in the development of tools.

At present, PROSE consists of three main components:
- **I_Prose**: interpreter for an extended Prolog (called **Env_Prolog**);
- **E_Prose**: Env_Prolog oriented editor;
- **D_Prose**: debugger.

The interpreter is written in C language, while other tools are all written in Env_Prolog. Our purpose is absolutely not that of building a complete and competitive programming environment. Our primary motivations to develop PROSE are rather the following ones:

- experimenting with usual interpreter implementation techniques in the special case of Prolog implementations;
- finding out possible extensions to standard Prolog which allows the language to better support the construction of programming environments (mainly in the direction of enhancing *metaprogramming* capabilities of Prolog);
- testing whether and how much techniques usually employed in the development of advanced tools for traditional programming languages are well suited to the special case of Prolog and, on the other hand, how Prolog special features influence and can be adequately exploited in the development of these tools.

2. An overview of I_Prose

I_Prose is the interpreter of the PROSE environment and its main component upon which all the other tools are based. The language implemented in I_Prose, named Env_Prolog, extends Prolog with new features which are mainly devoted to the support of the programming environment. More precisely, Env_Prolog extends the de-facto standard C_Prolog [15] with the following features:

- *infinite terms* [3];
- clauses and *programs as data,* and, as a consequence, the capability to have different programs in the program database at the same time;
- a number of *built-in* (meta-)predicates to handle terms and clauses;
- some mechanisms and built-in's for the *control of program execution.*

All these features will be discussed in more details in the next sections.

As regard to the implementation of I_Prose, we can mention the following features as the most distinguishing ones:
- a particular *unification algorithm* ;
- *uniform internal representation* of all terms and clauses of a program;
- derivation of the interpreter implementation by stepwise refinements of abstract formal specifications of an operational semantics of Prolog [11].

The unification algorithm used in I_Prose is an implementation of the algorithm proposed in [9] which in turn is a variation of the well-known algorithm by Martelli and Montanari [8], adapted to deal with infinite rational terms. This algorithm is based on the notion of *multiequation*, as a means to group together terms and variables which must be unified without having to perform any explicit substitution, and on some basic transformations on sets of multiequations which transform sets into equivalent ones [8,9].

Introduction of infinite terms is motivated firstly by interpreter's performance considerations (it is possible to suppress the occur-check operation still maintaining correctness) and, secondly, by the increased expressiveness of the language (cyclic data structures, e.g. graphs, circular lists, can be represented in a natural way). As an example, the program

```
eq(X,X).
?- eq(X,f(X)),eq(Y,f(Y)),eq(X,Y).
```

in Env_Prolog, has the solution:

```
{X,Y} = f(X)
```

whereas an implementation of Prolog with a standard unification algorithm loops forever trying to unify the infinite terms which are obtained from the first two subgoals in the given query (the result in the above example is a multiequation which expresses the fact that X and Y are equivalent and they are both bound to f(X)).

In the actual implementation of the unification algorithm, sets of multiequations are represented as *graph* data structures, which can be easily implemented in conventional imperative languages like C, by means of dynamic structures with pointers. Each multiequation corresponds to a different node in the graph (intermediate variables are introduced to have all terms with depth 1) and equivalent variables are linked together, with only the last one possibly bound to a non-variable term. (At this level our algorithm is very similar to the one proposed by Fages in [17]). For more details on the implementation of the unification algorithm see [16].

The use of this unification algorithm, in addition to the way I_Prose has been derived, that is by stressing similarities between interpreters for conventional programming languages and Prolog interpreters [11], has led to the choice of *"copying"* rather then *"structure sharing"* [12] to represent terms which are being unified.

On the other hand, the internal representation of source terms (i.e. not copied terms) has been designed in such a way to allow a *uniform view* of all internal data structures, without giving up the requirement of good execution efficiency and memory usage.

3. Programs as data

As we have already noted, the capability of a language to handle programs as data is a very valuable property in the construction of a programming environment. Standard Prolog implementations do not completely satisfy this requirement.

On the contrary, everything is considered as a term in Env_Prolog and can therefore be dealt with as data. In particular, a clause is a term with (infix) principal functor ":-", so that it is possible to write, for example:

```
    a :- b,c,d.
?- X :- Y,Z.
```

and have the answer:

```
{X} = a
{Y} = b
{Z} = c,d
```

where the given clause is considered as a fact rather than a conditional rule to be executed. In C_Prolog, on the contrary, the special built-in predicate *clause(H,B)* is required to deal with clauses as data and some limitations are imposed on the way its arguments can be instantiated (namely, H cannot be a uninstantiated variable) due to the ad-hoc internal representation of clauses.

Moreover, in I_Prose, also a whole *program* (that is a finite set of clauses) is considered as a term, with (infix) principal functor "|". For example, the program:

```
p.
p :- q.     corresponds to the term:
q :- r.                  |
r.                      / \
                 p:-true  |
                         / \
                    p:-q   |
                          / \
                     q:-r    |
                            / \
                       r:-true  {}
```

A special *syntactic notation* has been defined to represent a program in a more concise form (like with lists in standard Prolog)

```
{c1.c2. ... .cn}
```

where c1, c2, .. are clauses and {} denotes the empty program.
The internal data structures that represent programs as terms are exactly the same as those used for other terms. Therefore it is possible to work uniformly on programs as well as on other terms, without loosing any efficiency in accessing inner subterms. The interpreter is able to distinguish between programs and other terms and to build suitable data structures (e.g. indexes, ...) which facilitate an efficient execution of the program itself.

All predicate names in a program are local to that program and the interpreter keeps different indexes for different programs. The set of clauses in the program database can be partitioned into smaller separate subsets. To allow a program to refer to a different one, meta-predicates of standard Prolog (namely, call, clause, assert and retract) have been modified in such a way they can explicitly specify the program they work on, as an additional argument. For example we can have

1. ?- **ecall**(p(X),{p(X):-q(X). q(a)}).
   {X} = a

2. try(X) :- Y = {p. p:-q. q:-r. r}, **eclause**(q,X,Y).

```
?- try(X).
{X} = r.
```

The built-in predicate *ecall* allows a goal to be solved into any program which is visible from it. Notice that the standard built-in *call* is still used whenever the goal has to be proved in the current program. Predicates *eclause, eassert* and *eretract* are defined in a way similar to the corresponding C_Prolog built-in predicates. In particular, *eassert* and *eretract* operate by side-effects on the program they receive as argument.

Several programs can be simultaneously present in the program database and it is easy to associate a different symbolic name to each of them. For example, the following definitions

```
alfa mod {p. p:-q. q:-r. r}.
beta mod {r(X):-s(X),p. p. s(a)}.
```

where *mod* is a *user-defined* infix operator, can be interpreted as the definition of two programs named *alfa* and *beta* respectively (we'll refer to them also as *"module"* definitions). Thus it is possible to solve a goal like:

```
?- alfa mod X.
```

getting as its result that X is bound to the program named *alfa.*

Names can be used in combination with metapredicates which operate on programs, as in the following example:

```
modcall(G,N) :- N mod P, ecall(G,P).

?- modcall(p(X),alfa).
{X} = a.
```

The *mod* operator is user-defined and can be changed as one wishes. The only built-in which is aware of this operator is the modified version of predicate **consult**. It has the form

```
consult(prog_file,prog_name)
```

and the Prolog program in *prog_file* is loaded into the main memory with the assertion:

```
prog_name mod {"program in prog_file"}.
```

The program can be now referred using *prog_name* in the way seen above. If *prog_name* is omitted the name *user* is used as a default.

Modules has been widely used in the construction of PROSE. E_Prose, D_Prose and user programs are distinct modules. Programs to handle graphical output on the screen or to manage files within the environment are defined as inner modules.
Modules can be nested at any depth, since they are terms. A program can use any predicate belonging to any module defined within it, but it can not use predicates belonging to modules defined at the same or outer level. To allow a module *alfa* to be visible to another module *beta*, an outer program must explicitly pass *alfa* to *beta,* like for example in the following program

```
alfa mod { ... }.
beta mod { ... }.
export :- alfa mod X,
          beta mod Y,
          eassert( (alfa mod X), Y).
```

After *export* has been called, program *beta* is modified in such a way it contains a definition of the module *alfa* and it can now refer to any predicate in *alfa* through one of the above metapredicates.

Programs can be used also as a way to collect clauses defining some 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 pattern-matching). For example, the following two

clauses define a general program to find out a path between two nodes X and Y in a graph G:

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

where G is represented as a set of assertions (i.e. a program) like for instance in the goal:

```
?- p(a,d,{a(a,b).a(a,c).a(b,c).a(b,d).a(c,d)}.
```

Like with modules, it is also possible to associate a name to a graph. For example

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

and thus the above goal could be rewritten as

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

Notice that the notion of *module* in Env_Prolog is similar to Bowen and Kowalski's notion of *theory* [2]., with *ecall* corresponding to the *demo* predicate. On the contrary, our approach is quite different from those proposals, like for instance M_Prolog [1], where modules are special syntactic entities which allow the programmer to specify visibility rules of names within them.

The *copying* based technique used in I_Prose would require that when solving a goal like ?-alfa mod X a new copy of the whole program bound to X is made if the program contains any variable. To avoid this heavy operation, we assume that a program is always considered as a "ground" term. I_Prose can recognize a ground term and avoid to make any copy of it when the term has to be unified. More complex operations on programs as data (like for instance appending two programs) are still being investigated at present.

4. Interpreter and programming environment

As we have pointed out in the first section, implementation of the environment tools requires that the implementation language supplies a number of facilities which are usually not available in standard Prolog implementations.

In the project described in [5] these Prolog deficiences have been overcome by using an *internal representation* of programs in the form of Prolog assertions which are accessible to all the tools of the environment.
We also have followed this approach in the development of a first prototypical implementation of an Editor and a Debugger for the PROSE environment, both written in C_Prolog. The internal representation of programs we have used is illustrated by the following example:

```
p(X) :- q(X,Y),r(a).
        ⇓
iclause(Cr,Pr,Nr,p(_1),[q(_1,_2),r(a)],[v(_1,'X'),v(_2,'Y')]).
```

where the first three arguments are used by the Editor to move from one clause to another (Cr, Pr, Nr = current, previous, next clause reference, respectively). The last argument allows the interpreter to maintain the association between the internal name and the corresponding external one for each variable in a clause (since C_Prolog does not provide this facility).
A program is transformed into its internal representation by the Editor.

An advantage of using this intermediate representation is the separation between predicate names of user programs (represented as *iclauses*) and predicate names of tools (anyway separation among tools is still a problem).
The major drawback of this solution is the necessity to introduce a translation step which of course limits environment interactivity.

Another drawback of this solution is the difficulty and/or inefficiency to implement some operations of the environment in standard Prolog, especially if compared to the relative simplicity they could be directly implemented within the interpreter. For example the Editor in PROSE should perform lexical and syntactic analysis being a language oriented editor. Implementation of these operations in Prolog is quite cumbersome and inefficient. On the other hand the interpreter already executes such operations internaly and it seems reasonable them to be exploited.
In the same way, such operations as storing the symbolic name of variables or traversing clauses could be done directly by the interpreter with only a negligible overhead.
As regard to the Debugger, it can be written in Prolog as a *metainterpreter* which is able to execute the *iclause* internal representation of a program. However to have a really significant tool, such a metainterpreter must simulate most of the execution process of the interpreter, resulting in a complex tool, which causes a program under debugging to be executed very slowly.
Again if the user could access to some of the information the interpreter uses internally to control the execution of a program, implementation of the Debugger would be strongly simplified.

The approach we have followed in PROSE assumes that the interpreter (I_Prose) maintains informations which are useful for the construction of the basic tools of the environment, letting these informations be accessible to the user through some new built_in predicates. We'll briefly describe these new facilities in the next two sections.

5. Editor support

In this section we briefly describe the most important new built-in predicates which Env_Prolog supplies and which are mainly used in the development of the editing facilities of the PROSE environment. Usually they define operations which are difficult or highly inefficient to implement directly in Prolog.

Syntactic analysis and term construction

- **mkterm**(L,T,N).
  L is a list containing the ASCII representation of the term T. If the representation in L is not syntactically correct, N is an integer indicating the position in L where the first error has been found.
  Identifiers with capital initials are considered as constants in T.

- **mkvars**(T1,T2).
  T1 and T2 are the same term except for identifiers with capital initials which are considered as constants in T1 and as variables in T2.

Example:

```
?- mkterm([102,40,97,44,103,40,97,41,41],T1,_),
   mkvars(T1,T2).

   {T1} = f('X',g(a))
   {T2} = f(X,g(a)).
```

These two new predicates allow the Editor input phase to be strongly simplified. Syntactic analysis is performed by the *mkterm* predicate and can be applied to single predicates or to a clause as a whole. A clause can be built incrementally, adding new predicates or modifying existing ones. Predicate *mkvars* allows equal variable identifiers to denote the same variables within a clause when the clause is stored

in the program database even if it is constructed incrementally.
Having these predicates as built-in does not limit the operations the Editor can still perform. For example, automatic balancing of parenthesis, removing superfluous blanks in the input stream, error diagnostic messages and many other editing operations are all charged to the editor.

Clauses handling

Some new built-in predicates are defined in Env_Prolog to exploit the physical ordering of clauses. Clauses are identified by an internal unique identifier (Clause Reference) whose value has no meaning for the user. The interpreter keeps clauses ordered using the same data structures it uses for the internal representation of a program (that is a tree of "|" operators) without having to add any new structure.

- **assertp**(Cl,ClRef).
  Clause *Cl* is stored in the program database just before the clause identified by ClRef. Insertion is made in such a way to guarantee that clauses belonging to the same procedure are all necessarily contiguous.

- **adj**(ClRef1,ClRef2).
  ClRef1 and ClRef2 are references to two adjacent consecutive clauses. If ClRef2 (ClRef1) is the reference to a clause and ClRef1 (CLRef2) is bound to a variable, then *adj* can be used to obtain the previous (next) clause of the given one. The program d.b. begins and ends with two fictitious clauses with ClRef=0 and ClRef=-1, respectively. Thus *adj* can be also used to find the first and the last clause in a program.
  The following is an example which shows a typical use of *adj*.

  Example: *go to* the n-th clause.

```
goto(N,Ref) :- first(FCl),go(N,1,FCl,Ref).
go(X,X,R,R).
go(N,I,R,NewR) :- adj(R,Next),
                  I is I +1,
                  go(N,I,Next,NewR).
first(ClRef) :- adj(0,ClRef).
```

Symbolic variable names

For each variable in a term I_Prose preserves the user defined name in the internal representation of the term itself. The built-in predicates **write**(X) prints a term (including clauses and programs) with non-instantiated variables represented by their original symbolic names, contrary to what is done by usual C_Prolog implementations. It is evident the utility of this facility both to the Editor and the Debugger.

A problem arises with *renaming* of variables which is done by the interpreter whenever a predicate is unified with the head of some clause. I_Prose faces this problem by appending a univocal index to the name of renamed variables. For example, given the program

```
p(f(X),Y) :- q(Y).
q(f(X)).
?-p(X,Y).
```

we get the result:

```
{X} = f(X_1)
{Y} = f(X_2).
```

6. Debugger support

In C_Prolog and in many other Prolog implementations the Debugger is completely embedded within the interpreter. This solution assures good efficiency but no flexibility at all as regard to debugging policies. At the opposite extreme is the solution based on metainterpretation, we have already cited in Section 4.

The approach we followed to develop the PROSE Debugger (D_Prose) can be considered as intermediate between these two extremes. Indeed, D_Prose is written in Env_Prolog but execution of the program under debugging is completely carried on at the object level. What the user can do in Env_Prolog is to force the interpreter to call a user-defined procedure whenever a goal has to be solved, and to control its execution through a number of special built-in predicates.

More precisely. When user requests the activation of debugging mode (through the execution of the built-in predicate **dbgon**) the interpreter transforms the execution of each goal G into the execution of the user defined procedure **dbgenter**(G). This procedure completely defines the Debugger. Within this procedure the debugger designer can use two new built-in predicates to control the execution of the given goal G:

- **select**(G,ClRef)
- **exec**(G,ClRef).

Predicate *select* gets the reference of the first clause in the (current) program whose head unifies with the given goal G. If no such clause is found, *select* fails. Upon backtracking, *select* gets the reference of the next clause, if it exists. Predicate *exec* solves goal G using the clause specified by *ClRef.*
If G is a built-in predicate, *select* does nothing more than to bind *ClRef* to the constant *builtin* and *exec* performs a *call(G)* .
Notice that procedure *dbgenter* can be composed of different alternative clauses like any other Prolog procedure and backtracking applies as usual to them.

Selection of these two primitives is the result of a careful tradeoff between efficiency and flexibility. Their implementation is done in such a way to avoid execution overhead as much as possible. In particular, substitutions computed during the execution of *select* are stored into an ad-hoc internal data structure so that they have not to be recomputed when the corresponding *exec* is called successively (this structure is automatically removed on exiting from the *dbgenter* predicate).

Of course *select* and *exec* must be used with the due care. In particular, the argument *ClRef* of *exec* must be the reference of a clause previously selected by a *select* predicate on the same goal G (actually G has been inserted in *exec* just to report the new variable substitutions that are possibly created by the execution of clause ClRef).

The current substitutions (created by *select* and *exec*) can be obtained if necessary at any time, through the special built-in **curr_substs**(G,ClRef,S1,S2), where S1 and S2 are two lists of pairs of the form:

[s(x1, t1),..., s(xn,tn)]

which represent current variable substitutions for G and ClRef respectively (G is always instantiated by *curr-substs* to its initial value, the one specified in the *select* call).

Another interesting new mechanism supplied by I_Prose which has been used in D_Prose is the one provided by the **do-undo** built-in predicates. It is mainly intended to support the implementation of an undo facility of the debugger but it can also be seen as a generalized cut, whose effect is not limited

to the procedure in which it appears. More precisely, the execution of the built-in predicate *undo*, causes the present computation to fail, and backtracking to be activated. All possible alternatives between *undo* and the immediately preceding *do* are rejected. Backtracking stops at the first possible alternative (if it exists) preceeding the *do* predicate.
For example, given

```
p:-q,r,do,s.
q.
q:- qbody.
r.
s:-t,undo.
s.
t:- ... .
?-p.
```

after executing *undo*, control is passed to the second clause of the procedure *q*.
Predicate *qbody* can be defined for example as:

```
qbody :- undoIng,q.
```

where *undoing* is another new built-in which is true iff an *undo* has been executed but a subsequent *do* has not yet (it allows normal backtracking to be distinguished from backtracking due to an *undo*). In this case the final result is re-executing the piece of program between *do* and *undo* in the very same way.
Predicate qbody could contain also another *undo*. Nesting of *undo* allows a whole computation to be redone backward.

The *do-undo* mechanism has been used in D_Prose to implement an undo facility which allows the user to undo any previous request to the debugger, redoing the computation again if needed.

With these facilities and with few other primitive mechanisms it is possible to build powerful debugging tools directly in Env_Prolog, in the way one likes more, without having to relay upon decisions already made and frozen within the interpreter. At present, D_Prose provides only facilities for tracing program execution like those provided by the C_Prolog debugger, but more sophisticated facilities and user interfaces are planned for the future and should be easily implemented in Env_Prolog.

7. Future work

The implementation of I_Prose has been completed at present and it needs to be extensively tested. It runs on VAX 780 and SUN under Unix 4.2 and its performance is almost the same as that of (interpreted) C_Prolog. For the near future we have planned to build also a compiler for Env_Prolog based on the WAM. The main purpose of this should not be to obtain better performance than with the interpreter, rather to experiment with the implementation of the new facilities Env_Prolog supplies in the framework of the now standard WAM.

The implementation of E_Prose and D_Prose using Env_Prolog has to be completed in few weeks (at present only simplified prototypical implementations are available). Some other tools to be integrated in the PROSE environment are under development at present. Namely, a partial evaluator, a type checker and a user interface.

PROSE will be used also to host the tools for the construction of knowledge based systems we are developing in a parallel project [14].

Another interesting problem we have planned to face in PROSE is the support of the notion of *program library*. It requires to tackle problems like those of visibility and protection of predicates, efficient

loading and restoring of library procedures, etc. (see for example [6] and [7] p. 161). The notion of "module" of Env_Prolog should be used advantageously to face these problems.

Acknowledgments

We wish to thank all people who have contributed to the development of PROSE and in particular L.Arcostanzo, W.Manassero and G.Schmitz for their effective contribution to the implementation of I_Prose.
This work has been partially supported by MPI 40% project ASSI.

References

[1] J.Bendl, P.Koves, P.Szeredi: The MProlog System; in Proc. of the Logic Programming Workshop, (S-A.Tarlund ed.) Hungary, July 1980.
[2] K.A. Bowen and R.A. Kowalski: Amalgamating language and meta- language in logic programming; in Logic Programming, (K.L.Clark and S-A.Tarlund, Eds), Academic Press, 1982, 153-172.
[3] A.Colmerauer: Prolog and Infinite Trees; in Logic Programming, (K.L.Clark and S-A.Tarlund, Eds), Academic Press, 1982.
[4] W.F.Clocksin and C.S.Mellish: Programming in Prolog, Springer Verlag, Berlin 1981.
[5] N.Francez et al.: An Environment for Logic Programming; in Proc. of the ACM Sigplan Symp. on Languages Issues in Programming Environments; Seattle, June 1985, 179-190.
[6] A.Feuer: Building Libraries in Prolog; AAAI-83, August 1983, pp. 550-552.
[7] Kluzniak, Swpakozicw: Prolog for programmers; Academic Press, 1985.
[8] A.Martelli and U.Montanari: An Efficient Unification Algorithm; ACM TOPLAS, 4,2, April 1982.
[9] A.Martelli and G.F.Rossi: Efficient Unification with Infinite Terms in Logic Programming; in Proc. of FGCS84: International Conf. on Fifth Generation Computer Systems, Japan, 1984.
[10] A.Martelli and G.F.Rossi: Toward a Prolog Programming Support Environment (in italian); Proc. of the First National Conference on Logic Programming, Genova, March 1986.
[11] A.Martelli and G.F.Rossi: On the Semantics of Logic Programming Languages; in Proc of the 3rd Conf. on Logic Programming, London, July 1981.
[12] C.S.Mellish: An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter; in Logic Programming, (K.L.Clark and S-A.Tarlund, Eds), Academic Press, 1982, 99-106.
[13] A.Martelli and G.F.Rossi: PROSE: a Prolog Support Environment (in italian); Proc. of the Second National Conference on Logic Programming, Turin, May 1987.
[14] L.Console and G.F.Rossi: FROG: a Prolog-based system for Prolog-based knowledge representation; in Artificial Intelligence and Information-Control Systems of Robots-87, (I.Plander, ed.), North-Holland, 1987,179-183.
[15] C-Prolog User's Manual - Version 1.5; edited by F.Pereira, Technical Rept. 82/11, Edinburgh Computer Aided Architectural Design, Univ. of Edinburgh, February 1984.
[16] A.Martelli and G.F.Rossi: An implementation of unification with infinite terms and its application to logic programming languages; Technical Rept., Dipartimento di Informatica, Univ. di Torino, 1987.
[17] F.Fages: Formes canoniques dans les algebres booleennes et applcations a la demonstration automatique; These de 3eme Cycle, Universite Paris VI, June 1983.