

LOGOS: An Object-Oriented Scheme to Implement Logic Programming Languages

Liviu-Virgil Ciortuz, Mirela Petrea
Department of Computer Science
"A. I. Cuza" University of Iasi
6600 Iasi, ROMANIA

May, 1994

Abstract

This paper introduces LOGOS, an object-oriented framework to present and implement in a unitary manner different logic programming languages (abbreviated in the sequel as LPLs). LOGOS is made of

- an abstract¹ logic programming language (here refereed as AbstractLOGOS);
- a class library of main data structures and actors which serve for implementing LPLs as instances of AbstractLOGOS;
- an introductory methodology for LPLs implementation in LOGOS.

Finally, estimations for usefulness and efficiency of working in LOGOS are given.

0. Introduction

We present a new perspective for implementing LPLs. Concerning not only the "pure" level of LPLs but also usually not-addressed aspects like implementing incompletely specified LPLs, this perspective comes along with the *object-orientation* paradigm which proposes

- object classification into *classes* and
- class stratification into *hierarchies*, by *derivation* of one class from one or more classes;
- class/object characterisation through encapsulated *methods*, together with
- type/value/methods *inheritance* from one class to its descendants.

Object-oriented systems could also incorporate other features like

- defining *abstract classes*, which do not have any *instance* object on their own and serve only for deriving other - *real* - classes,
- possibly *overwriting* methods in derived classes, so cancelling the value inheritance mechanism, and
- using *virtuality* in their method definitions, which means that a function defined in a base class, when called - by inheritance - from a descendent class could recognise the level from which it was called, and communicate with objects on that level.

We will show that this perspective is quite useful for LPLs implementation, due to the attended benefits in object-orientation: code reusability and extensibility, rapid debugging, integrating and communication of more tools into a single system.

In the LOGOS scheme, code reusability is linked to the definition of an abstract logic programming language, here referred AbstractLOGOS. It incorporates much of what certain

¹ The "abstract" qualifier used here comes out directly from the object-orientation paradigm and says that the class it qualifies comes into reality not by instantiations but through derivation into other - real classes.

LPLs do have in common in their conception. (Note that here what is “common” is not meant to be “general”; exceptions could be defined by derivation and overwriting.) So, AbstractLOGOS concerns especially the “top” level of LPLs definition and implementation; particular details are left on the extensions behalf.

Any real logic programming language when implemented in the LOGOS scheme will become a derivation/extension instance of AbstractLOGOS. Only differences from the abstract language are subject for their implementation. Moreover, the LOGOS scheme is in itself easily refinable and extendible. So, new LPLs could be added to or derived from those already in. For instance, even we started our definition for AbstractLOGOS starting from what Proplog, Datalog and Prolog have in common, we are implementing now in LOGOS a “core” CHIP (a logic programming language with finite domain constraints, see [Hen,1989]), LOGIN (a logic programming language with “built-in” inheritance, see [AKN,1986]), and DFL (a frame-based logic programming language in the framework of F-logic, see [Cio,1994]).

Formal definitions for abstract logic programming languages in general, and for AbstractLOGOS in particular are given in Section 1. The overall implementation conception of LOGOS is given in Section 2. We conclude with Section 3 which presents details of real LPLs definition and implementation in LOGOS.

Finally, a question should be answered: Which are the benefits of using such a scheme for implementing LPLs? Our response addresses the following points:

- rapid development of tools for new LPLs, with comparably performances to the “dedicated” ones;
- incremental and compact presentation and/or learning of several LPLs into a single framework. (This is in fact the reason that firstly determined our work, together with the personal need to implement a new LPL without starting it directly from scratch, or destroying it by repeated changes while the definition refines.) Consider these differences when compared to [MW,1989];
- coexistence of several tools into a single one, with very reduced increasing of code. By consequence, we expect that intelligent choosing and run of adequate logic programming tools by code examination may lead to better performances;
- LOGOS could be also implemented in declarative languages (instead of imperative object-oriented languages like C++, in which we work now). An implementation of LOGOS in DFL, the frame-based logic programming language addressed above is in progress at our department. This implementation will lead to even a more compact code;
- last but not least, LOGOS’ novelty is perhaps the possibility to deal in a concrete and valuable manner with incompletely defined LPLs.

1. Introducing Abstract Logic Programming Languages

This section introduces a formal definition for abstract LPLs. Then it individualise among them the AbstractLOGOS language which is the core of our LOGOS system. We address the reader that some of the following definitions are informal ones to some extent, or more exactly: incompletely specified. (See terms like: rule, circumstance, completely). But you will see, this is the right manner to do what we intended to do. And you could see just from the sequel that equivalent formal definitions could be given, but this is not appropriate for the size of this paper.

Definition 1:

Let W be an *alphabet*, i.e. a non-empty countable set of symbols.

We call *language* upon W a subset L of $W^* = \cup W^n$, $n = 0, 1, \dots$, where W^n is the n -fold Cartesian product of W .

An *expression* in L is an element w from the set defining L .

A set S of rules which decides if an expression $w \in W^*$ belongs to L is called *syntax* of L .

A set T of rules which associates truth values (true, false) to each expression $w \in L$ (possibly under different circumstances) is called *semantics* of L .

Definition 2:

An *abstract logic programming language* is a tuple $L = \langle W, S, T \rangle$.

If the W , S , and T components of L are completely specified, we call L a *real* logic programming language.

So, an abstract LPL is a rather incompletely specified one. By consequence, no “real” program will be written and executed in an abstract LPL. However, it’s a mistake to consider that abstract LPLs are not useful. They could be fully implementable, as we will show in the case of AbstractLOGOS, and they could serve as a nice base for completely specifying ‘real’ LPLs, and this will be also shown in the sequel for Datalog, and suggested for Prolog, “core” CHIP. Partial implementation details for DFL in the LOGOS framework will be given in the last section. So, once an abstract LPL has been defined, a surprising number of real LPLs could be rapidly put to work in a modular manner.

Now we exemplify the above notions with **the AbstractLOGOS definition**. It is the abstract logic programming language which is the core of the present LOGOS implementation. We should say that it was obtained by analysing what is “common” to the definitions of well-known languages Datalog, Prolog and Proplog (see [MW,1988], [Llo,1987], and also [CKW,1993], for their ‘contextualized’ form).

- AbstractLOGOS alphabet:

the set of *parameter symbols* and *variables*, together with

a collection of *special symbols*: $(,), , , ., :-, ?$. (this is the ‘procedural’ version of the ‘declarative’ one: $(,), \wedge, \vee, \Leftrightarrow, \Rightarrow, \neg, \exists, \forall$).

Parameters and variables are finite sequences of letters (in the Anglo-Saxon alphabet), digits and underscore ‘_’, beginning with a lower case respectively an upper case letter.

- AbstractLOGOS syntax:

the set of the following context-free rules written in extended Bachus-Naur form:

```
<program> ::= <clause_list> <goal> EOF
<clause_list> ::= <clause> {<clause>}
<clause> ::= <atom> .
<clause> ::= <atom> :- <atom_list> .
<atom_list> ::=  $\epsilon$ 
<atom_list> ::= <atom> {, <atom>}
<term_list> ::=  $\epsilon$ 
<term_list> ::= <term> {, <term>}
<goal> ::= ?- <atom_list> .
```

Angle and curly brackets, and also the ϵ symbol in the above rules are meta-grammar symbols. Angle brackets surround non-terminal symbols, while curly brackets denote zero or more occurrences of the surrounded sequences of symbols. The ϵ symbol stands for the empty string.

Note that atoms and terms remain unspecified in the above grammar (but they could be completely specified in different extensions of it). Also, compound expressions commonly used in declarative semantics are not specified here, because we preferred to express a ‘procedural’ version needed for implementation. In the sequel the *expression* notion will address clauses, atoms and terms.

- AbstractLOGOS *declarative* semantics:

semantic structure (or, interpretation): $I = \langle U, I_s \rangle$, where U , a non-empty set, is the universe of interpretation, and I_s is (a mapping, or in general) a tuple of mappings (no more specified);

variable assignment in a semantic structure I : a mapping $as: V \rightarrow U$, where V is the set of variables in the AbstractLOGOS alphabet. Note that a variable assignment is generally

extended to all terms, but how to do it for AbstractLOGOS remain unspecified (correlated with, but not implied by, the fact that term syntax is not specified); *truth values* assigned to atoms in a given semantic structure I and a variable assignment in I remain unspecified (due to the same reason as above), but truth values assigned to compound expressions (here including Horn clauses) can be fully specified for AbstractLOGOS following the well-known way in the first-order predicate calculus. Note the difference from the above remarks: even a notion is not entirely specified for AbstractLOGOS, it could be used in a subsequent definition which may be in itself entirely specified.

- AbstractLOGOS *procedural semantics*:

substitution: set of pairs (here referred also as *unit substitutions*) var/term, in which term does not contain var, and different pairs have different vars;

substitution instance of an AbstractLOGOS expression through a given substitution σ : the expression obtained by replacing simultaneously each occurrence of a var from σ with the corresponding term.

substitution composition: easily definable;

unification of expressions: a mapping $\text{unify}: E \times E \rightarrow \{\text{false}, \{\text{true}\} \times S\}$, where E and S are the set of all expressions, respectively substitutions in AbstractLOGOS. This mapping is no more specified. If $\text{unify}(e_1, e_2) = (\text{true}, \sigma)$, then σ is called *unifier* of e_1 , and e_2 . The notion of *most general unifier* for two (or more) given expressions is naturally definable;

selection rule: a mapping which associates to a each goal $G = ?- A_1, A_2, \dots, A_k$ one of its component atoms;

inference rule: a mapping which associates to a each program (pair of goal and list of clauses) a (new) goal;

interpreter: a function which decides for a given program P if it is or is not unsatisfiable. We can fully define now such an interpreter for AbstractLOGOS, linking together the declarative and procedural semantics.

- A top-down interpreter for AbstractLOGOS:

Input: a program $P = \langle G, \{ C_1, C_2, \dots, C_m \} \rangle$ where $G = ?- A_1, A_2, \dots, A_k$ is a goal and C_1, C_2, \dots, C_m are clauses.

Output: "No", if P is unsatisfiable,
"Yes", and σ , a computed answer substitution, otherwise.

Procedure:

$\sigma = \varepsilon$ (the empty substitution);

if G is the null list

 return "Yes", σ ;

else

 let be A_i the atom given by the selection rule;

 if there is a clause C in P, written as $\text{head}(C) :- \text{body}(C)$. such that $\text{unify}(\text{head}(C), A_i) = (\text{true}, \theta)$, and

 this procedure applied to $P' = \langle G', \{ C_1, C_2, \dots, C_m \} \rangle$, with $G' = ?- (A_1, A_2, \dots, A_{i-1}, \text{body}(C), A_{i+1}, \dots, A_k)\theta$ returns "Yes", and σ' , then

 return "Yes", $\sigma' \sigma$;

else

 return "No";

Note that the above algorithm could be refined for instance into a depth-first, or a breadth-first one, depending on how the clauses C are tested and how the selection rule works on these clauses.

Now, we came to the point where we can specify different LPLs in the LOGOS scheme, and we do it by reference to the AbstractLOGOS definition, namely by completely specifying it. For exemplification we give the Datalog specification. (We would find interesting to introduce here

DFL, a novel data frame-based logic programming language working in the subset of F-logic (see [KLW,1990], [Cio,1994], [CP,1994]). It would show how LPLs languages much different than those from which LOGOS arose could be specified and implemented in LOGOS. Due to the limited size of this paper, we restrict ourselves to implementation details for DFL in the last section.)

Datalog definition (related to) AbstractLOGOS:

- syntax: *add* (to the AbstractLOGOS grammar) the following rules:
 - <atom> ::= parameter_symbol
 - <atom> ::= parameter_symbol (<term_list>)
 - <term> ::= variable
 - <term> ::= parameter_symbol;
- semantic structure: *define* I_S as the tuple $\langle I_C, I_P \rangle$, where $I_C: S \rightarrow U$, where S is the set of all parameter symbols in the Datalog alphabet, and $I_P = \langle I_P^{(0)}, I_P^{(1)}, \dots, I_P^{(n)}, \dots \rangle$, with $I_P^{(n)}: S \rightarrow 2^{U^n}$;
- extension of a variable assignment on terms: *define* $as(s) = I_C(s)$, for each parameter symbol s .
- atom truth value in (the “circumstances” given by) the semantic structure I and the assignment as :
 $p(t_1, t_2, \dots, t_n)$ is true iff the n -tuple $(as(t_1), as(t_2), \dots, as(t_n))$ belongs to $I_P^{(n)}$.
- unification on terms:
 - $unify(t,t) = (true, \epsilon)$, for any term t ,
 - $unify(X,a) = unify(a,X) = (true, X/a)$, for any variable X and term a different from X ,
 - $unify(s,t) = false$, otherwise;
- unification on atoms: $p(t_1, t_2, \dots, t_n)$ unifies with $q(s_1, s_2, \dots, s_n)$ iff p is q and every t_i unifies with s_i .

Now, “core” Prolog could be defined in a similar way to Datalog, or even by reference to Datalog, refining only the term definition and what does this further implies!

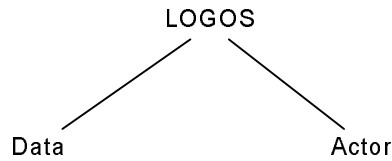
“Core” CHIP (see [Hen,1989]) is naturally definable by derivation from Prolog, i.e. adding finite domain variables to the alphabet, and extending unification on terms to finite domain unification. LOGIN (see [AKN,1986]) is obtainable from Prolog by introducing tags, and extending term definition and unification to ψ -terms.

All such languages that can be obtained from AbstractLOGOS by overwriting and/or further (completely) specifying its definitions are called here *instances* of AbstractLOGOS. (They form in our opinion a “LOGically Open System”, giving the acronym which identifies our scheme.) This AbstractLOGOS implementation is given in the next section, followed by implementation details for (some of) its instances in the last section.

2. LOGOS Implementation Conception

This section presents the overall implementation conception of LOGOS. The object-oriented programming language we used for the present implementation of LOGOS is C++ under the 3.1 Borland International release. An implementation of LOGOS in the object-oriented logic programming language DFL is in progress at our department.

From a particular point of view, LOGOS could be seen as a *class library*. Its root is an abstract class named LOGOS, and two sub-hierarchies, Data and Actor derive from it. The Data hierarchy contains classes defining main *data structures* and methods acting upon them, while the Actor hierarchy implements *actors* (like scanners, recognizers and parsers) dedicated to work on source logic programs.



We think the best way to present LOGOS class library is to provide it in an incremental manner:

- firstly, the main data structures hierarchy;
- secondly, logic programming concepts defined as classes refining those priority given; and
- finally, actor classes working on the source code of logic programs.

Note that the second point is the most important for the overall view on LOGOS. It provides in an implicit manner the implementation of AbstractLOGOS, including different interpreters in logic programming.

The class library for main data structures we have used in LOGOS implementation is defined by the hierarchy shown in Figure 1. It presents some similarities with Borland's C++ 3.0 class library, but is simpler, and dedicated for logic language implementation, including only those things working for this aim.

We did our best to make class names self-explaining. **Data** and **Container<T>** are abstract classes, i.e., there are no objects in these classes, they serve only as base classes for other (real) classes, containing only their main methods (type) declaration. <T> denotes a template (or: parameter) in the class definition (see [Bor,1992] for the template issue), and the lines correspond to the is-a (or: derivation) relation between different classes in the library. The is-a relationships dictate the inheritance of fields and methods in the hierarchy from top to bottom. Class names beginning with the letter I are specialised in working on pointers. This why their templates are of the form <T *>. A last explanation to be given concerns the difference we meant between the notions of sequence, vector, and array of objects. So, a sequence is a data structure that could contain a (not fixed) number of objects of the specified type. A vector stores n objects, while an array reserves storage space for n objects, and actually stores m, with $m \leq n$.

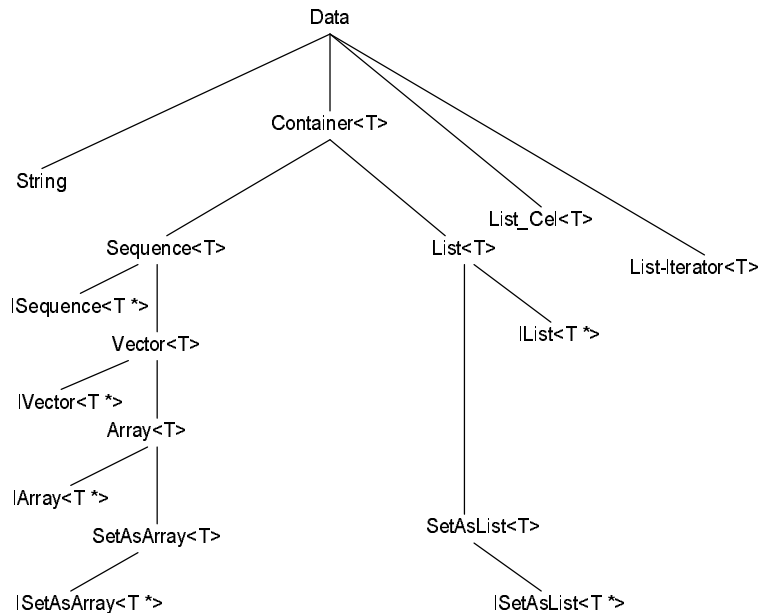


Figure 1: Main Data Structures in LOGOS

This hierarchy of main data structures will be used in the subsequent to define classes to implement specific concepts used in logic programming.

Many of these concepts are (derived from) instances of "templated" classes. For example, the class `LSubstitution` is derived from `ISetAsList<LUnit_Substitution *>`, which is an instance of `ISetAsList<T *>`, obtained for `T = LUnit_Substitution`. It follows that a substitution is (represented as) a set of pointers to unit substitutions. In the same way, a `LClause` will be seen as (derived from) a list of pointers to `LAtoms`. The class hierarchy of these logic programming concepts is given in the following figure. (The dotted lines correspond to templated class instances.)

We tried to present not only the derivation relationship between different classes, but also to respect the encapsulation of data in different classes. This is why in this diagram any class appears below all those classes from which it encapsulates data. (This is true for hierarchies presented in this paper.) For example, the class `Lazy_List` appears under the class `Lazy_Cell` because it is made of objects of the type `Lazy_Cell`. And this last one is below `LUnit_Substitution` because it encapsulates a pointer to `LUnit_Substitution`.

In the same way as we did for the previous class hierarchy, we've tried to make the logic programming class names self-explaining. We hope the single exception is the abbreviation `ST` (in `LST_Entry`) which stands for `LSymbol_Table`.

The beginning letter `L` comes from `LOGOS`. The classes so denoted are in fact those which implement `AbstractLOGOS`.

The only thing we should add is that the class `LProgram` contains the methods implementing different types of interpreters all using `SLD-resolution`. One of these interpreters is given in the Appendix at the end of this paper.

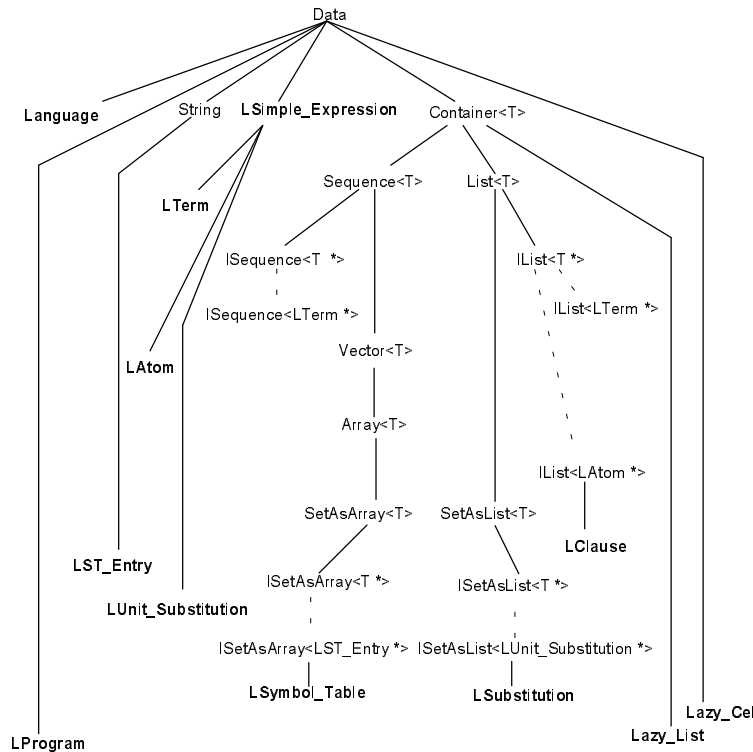


Figure 2: Logic Programming Concepts

The next diagram presents the hierarchy of actor classes working on logic programs up to obtaining their internal representation.

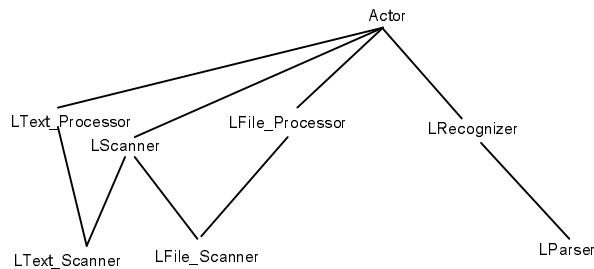


Figure 3: Program Actors in LOGOS

Actor is an abstract class from which are derived the other real actor classes in LOGOS. Its main task is that to define main error-handling methods.

LScanner is charged with the lexical analysis for source logic programs. So, it has to identify the lexical items (constants, functions, predicates, and variables) in the logic program (successively) loaded in its text buffer and to communicate them as an input to the syntax analysis. When associated with a file processor class, LScanner gives birth - by multiple inheritance - to the LFile_Scanner class which does the entire lexical analysis for logic programs saved on text files. LText_Scanner was designed to do the same task, but for logic programs loaded directly from scratch.

LRecognizer's methods validate in a top-down manner the syntactic correctness of the submitted logic program, according to the logic language grammar. The program representation structure is created by the LParser action, if the program was proven syntactically correct.

The main classes involved in the logic program representation are LProgram, LSymbol_Table, LClause, LAtom, LTerm, (the last two being derived from LSimple_Expression). All these classes are in fact abstract class. They don't have any physical realisation on their own (neither do they contains storage details in their definition), but they serve both to define the way LParser acts through its methods and to derive actual representation classes (with storage details) for definite programs in real logic languages. (For more specific details, see the last section.)

The LProgram encapsulates the representation obtained for the given logic program and defines different methods for goal solving: a top-down/depth-first interpreter, a top-down/breadth-first interpreter, a bottom-up interpreter, and a top-down/depth-first interpreter using lazy lists.

So finally a logic program (representation) is created by an LProgram constructor, and its goal is solved by sending a message corresponding to the chosen interpreter.

The next section shows the way classes presented above can serve to extend and overwrite (any part of) the definitions for AbstractLOGOS, in order to implement new LPLs.

3. LPLs Implementation in LOGOS

This section presents how to expand and overwrite AbstractLOGOS (constituent classes/encapsulated methods) to implement other LPLs.

Generally speaking, abstract classes implementing logic programming concepts (which are in fact "schematic" ones) have to be further refined - and we suggest to do it by templated derivation - into other (real) "language oriented" classes. As we said in the introduction, the first classes say you "how" to proceed, the last ones specify exactly "on" what object instances your work has to be done. This idea is illustrated by the following diagram which records our experience with implementations done for Proplog, Datalog and Prolog, and currently used for LOGIN and DFL.

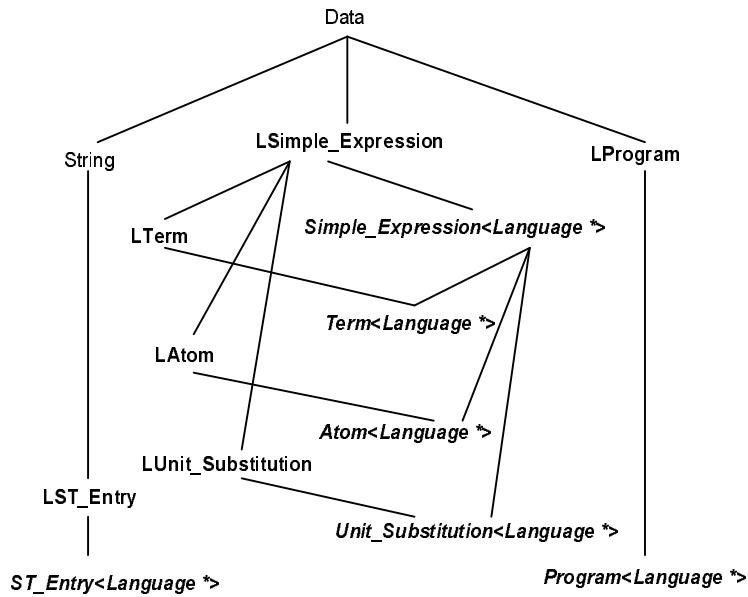


Figure 4: Specific Concept Classes for LPLs Implementation

Note that the (address of a particular) language name is used to identify (as template value) the classed particularly used to implement that language. Also, if for instance Prolog would be defined by reference to Datalog instead of AbstractLOG, (and in fact this will be the case for DFL), then Prolog specific concept classes will change (part of) their parents and method composition.

The use of templates (for the classes here introduced) as pointers to Language instances is not compulsory; it is just enough for uniform treatment of different LPLs implementation, and to increase code visibility and understanding. If someone wants to implement a single logic language within the LOGOS scheme, then he/she could simply renounce to template these classes. (However, the use of templates for class definitions in LOGOS could in no way be underestimated.)

In order to reach the representation level for logic programs written in a particular LPL, we use also specific actors as shown next.

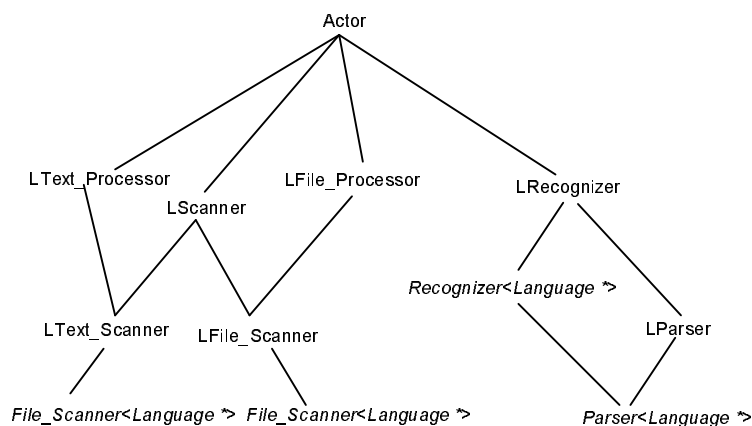


Figure 5: Specific Actor Classes for LPLs Implementation

Now we can give now the **Datalog implementation details in LOGOS**, corresponding to the definition (related to AbstractLOGOS) in Section 1:

- Create Datalog as object in the class Language by declaring

Language Datalog(Datalog_SpecSyms);

where Datalog_SpecSyms defines the special Datalog symbols set: :-, ,, ,, ?-, (,).

- Define Atom<&Datalog> and Term<&Datalog> classes corresponding to the non-terminal symbols added by the Datalog specific rules in Section 1. Encapsulate and define corresponding unifying methods (see function matches in Appendix B).
- Encapsulate in the Recognizer<&Datalog> and Parser<&Datalog> (to be defined previously) classes methods for analyse/represent Datalog atoms and term according to the rules in Section 1.

Note that no redefinition is needed for Datalog interpreters, since AbstractLOGOS interpreters are enough general to manage (through virtuality!) logic programs in Datalog, Prolog, Proplog, LOGIN ‘core’ CHIP and other LPLs.

Appendix B contains the about half of the Datalog implementation in LOGOS, namely the Atom<&Datalog> part. (The Term<&Datalog> part is even simpler.) Full Datalog implementation took about 150 lines of code. Implementation of Prolog is quite similar, and it is even more compact when done it wrt Datalog. So, one could note the reduced amount of text needed to obtain logic programming tools using LOGOS. Our claim (remaining to be proved) is that these tools, comparable in compactness with mete-interpreters written in some existing LPL, are more efficient, being much more linked to dedicated implementations.

As we have promised, we give finally some implementation details for DFL, the data frame-based LPL which is under development at our department. (This implementation is designed wrt Datalog).

- Create the DFL Language object by declaring

```
Language DFL (:-, ,, ,, ?-, (, ), @, :, [, ], ->, =>, ->>, =>>);
```

- Extend term definition to ‘methods’, using the following rules:

```
<atom> ::= <is_a_atom>
<atom> ::= <equation>
<atom> ::= <DF_atom>
<is_a_atom> ::= <term> : <term>
<equation> ::= <term> = <term>
<DF_atom> ::= term [ <method_list> ]
<method_list> ::= ε
<method_list> ::= <method> {; <method>}
<method> ::= <functional_method>
<method> ::= <fun_typing_method>
<method> ::= <set_val_method>
<method> ::= <set_typing_method>
<method> ::= <predicative_method>
<functional_method> ::= <term> <context> -> <term>
<fun_typing_method> ::= <term> <context> => <term_list>
<set_val_method> ::= <term> <context> ->> <term_list>
<set_typing_method> ::= <term> <context> =>>
<predicative_method> ::= <term> <context>
<context> ::= ε
<context> ::= @ <term_list>
```

Methods corresponding to these rules have to be encapsulated into Recognizer<&DFL> and Parser<&DFL> (to be defined previously), and classes for non-terminals in these rules should be defined, encapsulating unification methods (see [CP,1994]);

- Define ST_Entry<&DFL> such to include not only headed clauses, but also is-a relationships and conditions on these;
- The top-down inference procedure must be overwrite so to incorporate new inference rules: “late” resolution, factoring, is-a transitive and reflexive closure, is-a antisimmetry, merging, elimination (see [CP,1994]).

Conclusions and further work

This paper presents the object-orientation paradigm incorporated into a framework used to implement logic programming languages. Benefits of this idea are outlined: rapid expandability, modularity, code compactness, easily LPLs presentation and learning. A new implementation for the LOGOS framework is in progress at our department, now using an object-oriented logic programming language as declarative background.

Acknowledgements

We thank the students in the 1991-1992 and 1992-1993 courses on software engineering who motivated and encouraged us in doing this work. We are indebted to Mihaela Armănaşu who tested some functions in the scanner and recognizer class definitions.

Special thanks of Liviu Ciortuz go to his friends Dan Manastireanu, Robert McCuiston, Jerry Little and Sebastian Thaci for the computing environment they kindly offered him while this work was in progress, and to Iosif Stefanuti for providing access to his photocopy machine. Last but not least, thanks to Angela Ciortuz for constant waiting for him.

Thanks for the kind useful remarks of the anonymous referees of the 6th Workshop for Logic Programming Environments at ICLP '94.

Thanks also to all those who, even they didn't know, encouraged us in doing this work.

Bibliography

- [AKN,1986] H. Ait-Kaci, R. Nasr, *LOGIN: A Logic Programming Language with Built-in Inheritance*, Journal of Logic Programming, 1986:3, pp185-215.
- [Bor,1992] Borland Corporation, *C++ 3.0 Programmer's Guide*, 1992.
- [Cio,1994] L. V. Ciortuz, *DF-Logic: A Significant Subset of F-Logic*, (to appear) in Proceedings of TAINN III, The Turkish Symposium on Artificial Intelligence and Neural Networks, Ankara, 22-24 June 1994.
- [CKW,1993] W. Chen, M. Kifer, D.H.D.Warren, *HILOG: Higher Order Logic Programming*, Journal of Logic Programming, 1993:15, pp185-215.
- [CP,1994] L. V. Ciortuz, M. Petrea, *Formal specifications for DFL, A Data Frame-based Logic Programming Language*, in Proceedings of the Development & Application Systems Symposium, Suceava, Romania, 25-27 May 1994.
- [FLB,1990] C.N. Fischer, R.J. LeBlanc Jr., *Crafting a Compiler in C*, The Benjamin/Cummings, 1990.
- [Hen,1989] P. van Hentenrick, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.
- [Hol,1989] S. Holldöbler, *Foundations of Equational Logic Programming*, Springer-Verlag, 1989.
- [KLW,1990] M. Kifer, G. Lausen, J. Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Technical Report 90/14, SONY at Stony Brook, 1990.
- [Llo,1987] J. Lloyd, *Foundations of Logic Programming*, 2nd extended edition, Springer Verlag, 1987.
- [MW,1989] D. Maier, D.S. Warren, *Computing with Logic*, Benjamin/Cummings, 1988.


```

virtual void apply(LSubstitution &);
virtual LAtom * copy(void)
    { return new Atom<&DataLog>(*this); }
virtual boolean matches(LAtom *, LSubstitution &);
virtual LUnit_Substitution *findFirst(LAtom *, outcome_type &);
virtual void print();
friend Parser<&DataLog>;
};

void Atom<&DataLog>::rename_vars(LSubstitution & aSubst)
{
    if(args != NULL) {
        int theArity = getArity();
        LTerm * currentArg;
        token_type currentType;
        Unit_Substitution<&DataLog> * anUnit;
        for (int i=0; i<theArity; i++) {
            currentArg = (*args)[i];
            currentType = currentArg->getType();
            if ( currentType == variable ) {
                if( !aSubst.contains(*currentArg) ) {
                    anUnit = new Unit_Substitution<&DataLog>(*currentArg, st->nr_virtual_entries++);
                    aSubst.addAtTail(anUnit); }
            }
            else
                if (currentType == functional)
                    currentArg->rename_vars(aSubst); }
    }
}

void Atom<&DataLog> :: apply(LSubstitution & aSubst)
{
    if ( args != NULL ) {
        int theArity = getArity();
        for ( int i=0; i<theArity; i++ ) {
            if ( (*args)[i]->getType() == variable ) {
                LTerm * termDeref = aSubst.deref ( (*args)[i] , NULL);
                if (termDeref != (*args)[i]) {
                    delete (*args)[i];
                    (*args)[i] = termDeref->copy(); }
            }
            else
                (*args)[i]->apply(aSubst); }
    }
}

boolean Atom<&DataLog>::matches(LAtom * anAtom, LSubstitution & aSubst)
{
    if (*this != *anAtom)
        return false;
    outcome_type oc = none;
    LUnit_Substitution * lus;
    while (true) {
        lus = findFirst(anAtom, oc);
        if( oc == none )
            return true;
        if ( oc == clash )
            return false;
        if( lus->violates() )
            return false;
        aSubst.compose(lus);
        delete lus;
        apply(aSubst);
        anAtom->apply(aSubst); }
}

LUnit_Substitution * Atom<&DataLog>::findFirst(LAtom * anAtom, outcome_type &oc)
{
    LSequence<LTerm *> * argsToMatch = anAtom->getArgs();
    LUnit_Substitution * anUnit = NULL;
    if (args != NULL) {
        int theArity = getArity(), oc = none;
        for (int i=0; i < theArity && oc == none; i++)
            anUnit = (*args)[i]->findFirst( (*argsToMatch)[i], oc ); }
    return anUnit;
}

```

```
void Atom<&DataLog>::print()
{
    Simple_Exp<&DataLog>::print();
    if (args != NULL)
        args -> print(getAriety());
}
```