South African Computer Journal
Number 23
July 1999
Guest Editorial

Computer Science and Information Systems:
The Future?

Philip Machanick

Department of Computer Science, University of the Witwatersrand, South Africa
philip@cs.wits.ac.za

1 Introduction

As president of the South African Institute for Computer Scientists and Information Technologists (SAIC-SIT), I have visited a number of campuses and companies, in an attempt at arriving at a general assessment of the state of our subjects in South Africa.

An issue which I consistently pick up is that while everyone seems to think that computer-related skills are extremely important and in short supply, our academic departments are also extremely underresourced.

At the last Southern African Computer Lecturers Association (SACLA) conference (28-29 June, Golden Gate), I had the opportunity to discuss the problems other academics see. This editorial lists some of the problems reported at SACLA, and proposes a way forward.

2 Problems

At SACLA, I led a discussion of problems seen in our academic departments.

There was wide agreement that both Computer Science (CS) and Information Systems (IS) departments were under pressure to increase student numbers (massification), and were seen as cash cows to prop up less popular subjects. It was broadly agreed that staffing was a critical issue: too few posts for the workload, salaries way out of line with industry (half or less, as compared to the US, where an academic salary may be 80% of an industry salary). Recent graduates often make more than professors which makes it hard to persuade our students to become academics (even to do higher degrees). Attracting a recent PhD with a sense of adventure is possible, but attracting experienced people used to earning a salary in a strong currency is hard. IS jobs are worse than CS, as the skills required are more like those in business. Support staff salaries are an even harder issue: their skills relate even more directly to job descriptions in industry.

A problem in addressing our concerns is that we are so overworked that we don't have time for "politics": academics with no students have time on their hands, but we don't. More industry support not only with directly addressing problems but with taking on university administrations would be useful, but they too have major problems and don't have free time.

3 Solutions?

Solutions are harder to identify than problems.

The SACLA session ended with a proposal that we conduct surveys of our institutions and businesses, to find out what the problems are, as a starting point for going to university administrations, government and business.

Another idea was to attempt to find common cause with business in taking on problems they have in common with academia, including the skills shortage, the insufficient capacity of our education system, and dealing with employment equity.

One of our biggest difficulties is to free up time to deal with issues such as resource allocation within our universities. The "competition" is frequently other academics with time on their hands, since they have too few students, and therefore are in a position to spend time looking after their interests.

What is needed now is some thought about how to pull ourselves out of the mess we are in. In particular, we need strategies to exploit our strengths: our high demand among students, the high demand for the skills we produce and the ubiquitous applicability of computer technology.

Given the wide use of computers, it would seem obvious that our areas should be strongly supported by a range of role players, yet the fact that so many different groups are interested in computer technology in one way or another has tended to fragment efforts to enhance our industry and academic institutions.

Clearly, from conversations I have held, some departments are in much better shape than others. Even so, some kind of collective effort is likely to achieve more results than if we allow ourselves to be pushed around as individuals. Addressing the fragmentation of efforts seems a worthy goal in itself, to reduce duplication and contradictory goals.

I appeal to anyone who has constructive ideas on how to take our subjects forward to contact me. Let us work on building ourselves up. The economy depends on us, much more than on most other academic disciplines. It's time we made that point, and made it strongly.
SAICSIT’99
South African Institute of Computer Scientists and Information Technologists

Annual Research Conference 17-19 November 1999

Prepare for the New Millennium

Is there life after y2k?

Mount Amanzi Lodge, Hartebeespoort
near Johannesburg and Pretoria

keynote speaker: Barbara Simons, ACM President
and many other local and international speakers (academic and industry)

Call for Participation
papers in a many areas of Computer Science and Information Systems are expected
Price Waterhouse Coopers prizes: Best Paper R10000 • Best Student Paper R5000
please check the conference web site for accepted papers:
http://www.cs.wits.ac.za/~philip/SAICSIT/SAICSIT-99/

To Register

go to the conference registration web page:
or contact SAICSIT’99 Secretary for details:
Department of Computer Science, Senate House 1137
University of the Witwatersrand
Jorissen Street
Wits, 2050
South Africa
phone (011)716-3309 fax 339-3513 (international: replace 011 by 27-11)
saicsit99-info@cs.wits.ac.za

Dates and Publication Details
early booking deadline: 14 September • on-site registration starts: 17 November 1999
workshops, tutorials 17 November 1999 • paper sessions: 18-19 November 1999
papers will appear in a special issue of South African Computer Journal

sponsors

[Logos of sponsors]
A declarative and non-determinist framework for Dynamic Object-Oriented and Constraint Logic Programming

H. Abdulrab, M. Ngomo and A. Drissi-Talbi

Advanced Computer Sciences Laboratory PSI-LIRINSA, INSA de Rouen, France,
{abdulrab,ngomo,drissi}@lirinsa.insa-rouen.fr

Abstract

LOP (Logic, Object, Parallelism) is a system that integrates object-oriented and constraint logic programming. It is entirely designed and implemented, using object-oriented methodology, under the form of hierarchical levels of distinct natures. In this paper, we restrict our presentation to the first two levels of LOP that integrate declarative programming with object-oriented programming. We show how this fusion is done in LOP using, from one hand, freezing mechanisms, and from another hand, the two new notions: multiple states object and multiple definitions method. The consequence of this approach is that any message passing which appears in a request is interpreted independently from the control strategy. In particular, the reduction of a message passing does not depend on its position in the request. In this paper, we particularly show how the declarative programming is preserved in the presence of state changes and dynamic method redefinition. The bases of our implementation are finally detailed.

Keywords: multiple states object, multiple definitions method, freezing mechanisms, object-oriented programming constraint and logic programming, state changes

Computing Review Categories: D.3.1, D.3.3

1 Introduction

Since the last decade, several approaches have been realised in order to integrate the concepts of object-oriented programming (OOP) in the framework of Declarative Programming (DP). Many of these approaches gave rise to programming languages that merge OOP with logic programming (LP). The experiences of these languages show that the integration of the structural aspects (class and instance definitions, etc.) of OOP in LP can be done without major difficulties, while the integration of dynamic aspects (state changes, dynamic methods, etc.) in declarative logic programming is much more hard.

The language LOP [1, 10, 2] described in this paper aims at the integration of OOP with LP. The combining of these two paradigms is progressively realised and allows the separation among incompatible programming mechanisms of OOP and LP. Figure 1 presents the four first programming levels of LOP: NP1, NP2, NP3 and NP4.

These levels are organised and implemented in a hierarchical manner: each level inherits the maximum of characteristics of its previous ones and presents in a minimal way its own characteristics.

The level NP1 is a system of declarative programming integrated with static object-oriented programming (without state changes).

The level NP2 enhances NP1 by some dynamic aspects of OOP and preserves, in the same time, the declarative programming framework.

In NP3 level, the procedural programming aspects are introduced.

And, finally, some solvers of constraints (linear constraints on naturals, etc.) are added in the fourth level NP4, in order to use LOP for discrete application (discrete simulation, etc.) [2, 3].

The two main advantages of this hierarchical construction are both to allow the user the choice of the suitable programming framework for his application, and the good extendibility of such hierarchical construction.

This paper is devoted to the first two levels of LOP which ensure a declarative programming framework. Our goal is to describe a way that integrates dynamic aspects of OOP in the declarative programming framework. Our approach, based on freezing mechanism and multiple states object, will be detailed here and compared with some hybrid languages.

Recall first that in declarative programming, a program describes the related knowledge to a given application, and a general interpreter is devoted to the program execution. In this context, the different parts of the program has no operational effect and there is no hypothesis about the way they are used [16]. LP offers such a declarative framework: a program describes the logical rules of the problem without any hypothesis about the process resolution.

When the dynamic aspects of OOP (such as state changes, dynamic methods, etc.) are used, the preservation of the declarative framework becomes very difficult. Informally, consider the message obj< -meth1(...) sent to the object obj, where the corresponding method is meth1. When state changes and dynamic definitions method are authorised, such message will be interpreted relatively
to the used definition of the method meth1 and the current state of obj. For example, if another method meth2 changes the state of obj, then the result of the execution of the two messages obj ← meth1(...) and obj ← meth2(...) depends on the order in which these two messages are executed. More precisely, if meth1 depends on the state of obj, it is possible to have a different result if the order is inverted.

With such state changes, the user must take into account the order of messages sending. The knowledge expressed by the program becomes related to the manner in which it is executed. Thus, the declarative programming aspects are lost. Similar effects can be obtained when the dynamic definitions method are authorised.

The two first levels of LOP allows the message passing to be interpreted regardless of this order in the program. Thus, the declarative programming aspects can be preserved when state changes and dynamic definitions method are used.

2 NP1 level : Static logic and object-oriented programming

A LOP program is written by means of the three primitives: defclass, definstance and defmethod. These primitives are translated by a pre-processor integrated in LOP in order to create all the classes and instances structures of the program, as well as the rules base corresponding to the methods (see figure 2).

The goal of the pre-processor is to transform any program written with the three primitives defclass, definstance and defmethod into the rules base which can be directly used by LOP’s interpreter. The generated base can be seen as an interface allowing the access to the states of the created objects. Each rule body is composed of literals that can be either a message sent to an object, or a predicate call. In the case of message sending, the receiver is represented by its identifier which is already defined by the user.

2.1 Class, instance and method notions

LOP’s model is based on classes and multiple inheritance. The definition of a class can be done by the primitive de-class as following:

```
defclass(cla (s1 ... sm) (at1 ... atn));
```

where cla is the identifier of the defined class, (s1 ... sm) is the list of its direct super-classes and (at1 ... atn) is the list of its attributes names.

An instance obj of the class cla is defined by means of the primitive de-instance as follows:

```
definstance(cla obj (at1 vall) ... (atn valn));
```

where the pair (ati vali) corresponds to the attributes and their associated values.

The instances communicate by message sending. The reduction of a message depends on the appropriated method related to the receiver of the message. The definition of a method is done by at least one rule, called object rule. An object rule is composed of a head literal, and a conjunction of the literals that define the body of the rule. Each literal can be a predicate call or a message sending in the form:

```
0 <- meth(...).
```

The definition of the method meth over the class cla is done by the primitive defmethod as follows:

```
defmethod(cla (rule1) ... (rulen));
```

where each rule rulei has the form:

```
0 <- meth(X1 ... Xn) :- body.
```
The head O< -meth(X1... Xn) is an object literal whose a receiver is a logical variable. The variable O both allows to indicate the message receiver and plays the role of the self variable in some languages as Smalltalk-80 or Objective-C.

The use of the three primitives defclass, definstance and defmethod is illustrated by example1.

Example1

The program Prog I describes the definition of the class triangle with the attributes ab, bc and ac that represent the lengths of the sides of the triangle. equilateral(X), isosceles(S) and reference() are three methods defined over this class. equilateral(X) determines if the triangle is equilateral (X is the length of its sides), isosceles(S) determines if the triangle is isosceles in S (a vertex of the triangle) and reference() tests if the triangle is equilateral and each one of its sides has a length equal to 1. Finally, the method get, used in prog 1, is a predefined method which allows the object's state to be consulted. The literal obj< -get(ati X) allows the logical variable X to be unified with the value of the attribute ati relatively to the object obj. The method get can also be used to extract more than one attribute values as following: obj< -get(at1 X1 at2 X2 ...).

/* Definition of the triangle class */
defclass(triangle () (ab be ac) );

/* Creation of two triangles */
definstance(triangle tril (ab 1) (ae 1) (be 1));
definstance(triangle tri2 (ab 2) (ac 2) (bc 1));

/* triangle's methods */
defmethod(triangle
(O<-equilateral(X) :-O<-get(ab X ac X be X)));
defmethod(triangle
(O<-reference() :-O<-equilateral(1)));
defmethod(triangle
(O<-isosceles(a) :-O<-get(ab X ac X))
(O<-isosceles(b) :-O<-get(ab X bc X))
(O<-isosceles(c) :-O<-get(ac X bc X)));

Prog1- Definition of triangle class in LOP.

2.2 Reduction of message sending and freezing mechanism

The resolution schema of LOP is an adaptation of the Prolog's one to an object-oriented paradigm context. As mentioned before, a request is a conjunction of literals that can be either predicate calls or message sending. The reduction of a predicate call is realised in the same manner as Prolog. The reduction of a message sending obj< -meth(X1 ... Xn) is realised by using all the rules that define the method meth over the class of obj. The reduction of obj< - meth(X1 ... Xn) related to an object rule in the form O< -meth(Y1 ... Yn) :- body, consists of the unification of obj with O and Xi with Yi for all i. If such a unification succeeds, the body of the rule replaces the re-
Figure 3: c3 inherits m1 from c1 and m2 from c2

duced literal of the current resolvant, and the resolution continues in the new unification environment. Backtracking allows the remaining rules to be applied. Finally, the initial request succeeds whenever the resolvant becomes empty.

A message is called anonymous, if its receiver is a non instantiated logical variable (i.e. not bound to an object identifier). Such a message sending is delayed in LOP until the logical variable becomes bound, at that moment the delayed literal is reduced. In the following request, the message O<-equilateral(X) is firstly delayed, and reduced after the unification O = tril.

?- O<-equilateral(X), O = tril.
O = tril, X = 1.
success.

2.3 Default inheritance and explicit designation of the class

Methods and attributes can be either locally defined in the class or inherited from an ancestor class. The possible conflicts related to the multiple inheritance are solved by the computation of a precedence list for each class, that specify an order over its ancestors classes. The algorithm used in LOP to determine such a precedence list is that of CLOS [13].

This solution is adopted by default, and can be unsuitable for the user in some cases [22, 23, 24]. For example, consider 3 in which the class c3 has two ancestors classes c1 and c2, and suppose that the user wants to inherit the method m1 from c1 and the method m2 from c2. By default, c3 inherits m1 and m2 from c1, because this class is more specific than c2. Consequently, the priority rule avoids m2 of c2 to be inherited. An expensive solution would be to rewrite over c3 the rules that define m2 over c2, but this is clearly not optimal from a reusability point of view.

To solve this problem, we introduce in LOP the notion of the resolution class of a message. The definition of the method used for the message will be taken precisely from this class.

In a message of the form obj<-meth(...), the resolution class is the class of the receiver obj. To avoid this default solution, one can specify in LOP the desired resolution class as follows: C@O <-meth(...), where C is chosen to be the resolution class. In this case, the reduction is done by all the rules that define meth over C. If C is an unbound logical variable, the reduction of such a literal is delayed until the logical variable becomes bound, as for the anonymous messages.

This explicit designation has some advantages. The first one is related to the conflict resolution associated with multiple inheritance. The case of figure 3 can be solved by the definition:

(defmethod c3 {O$<-m2(...) :- c2O$<-m2(...)});

The method m1 is inherited, by default, from C1 and m2 is explicitly inherited from C2. In addition, the explicit designation of the resolution class allows some new redefinition of methods to be written by using different methods of different classes. Consider, for instance, the case of the method m3 defined over both C1 and C2. The definition:

(defmethod c3 {O<-m3(...) :- c1O<-m3(...)}
{O<-m3(...) :- c2O<-m3(...)})

expresses that m3 is defined over C3 by means of the rules of C1 and C2, in the same time. Such a use provides the same behaviour with monotonic multiple inheritance as in ESP [8] and POL [11]. Finally, this can offer the same features as the auxiliary methods of some languages as ESP and CLOS [13]. In this case, the definition of a method over a class can call and enhance some definitions of other methods defined over other classes. Here is an example where the method reference() is redefined over the class triangle_colour (see prog2).

Example 2

/* Definition of the class triangle_colour */
defclass(triangle_colour (triangle) (colour));

/* Creation of two coloured triangles */
definstance(triangle_colour tri_coll
(ab 1) (ac 1) (be 1) (colour white));
definstance(triangle_colour tri_col2
(ab 1) (ac 1) (bc 1) (colour blue));

/* redefinition of the method reference */
defmethod(triangle_colour
{O<-reference() :-
triangle@O<-reference(),
O<-get(colour white)});

Prog2- Definition of the class triangle_colour.

The class triangle_colour inherits from the class triangle its methods and its attributes. It has an additional attribute which corresponds to the triangle’s colour. The redefinition of the method reference() is based on its initial definition over the class triangle and on the additional condition that specifies that the triangle should be white. In the following requests:

?- tri_coll<-reference().
success.
Example 1 (continued)

?- tri1< -isosceles(A).
  A = a.
  success.
  A = b.
  success.
  A = c.
  success.

?- tri2< -isosceles(A).
  A = a.
  success.

?- tri1< -equilateral(X)
  X = 1.
  success.

?- tri1< -reference()
  success.

?- tri2< -isosceles(A).
  A = a.
  fail.

?- tri1< -equilateral(X)
  fail.

?- tri1< -reference()
  fail.

?- tri_col2<-reference().
  fail.

?- triangle@tri_col2<-reference().
  success.

the triangle tri_col2 is not considered to be a reference (because it is blue). But in the third request, it is considered to be a reference, because the definition of reference (which does not take the colour in account) of the class triangle is used.

3 NP2 level: multiple states in logic and object-oriented programming

3.1 Multiple states object

In LOP, an object has an incomplete structure that can be extended or reduced during the resolution phase. This structure allows the registration of the initial state of the object (i.e., the set of the initial values associated with the object attributes). When the object state is modified, the object structure allows the new state to be memorised. But, each added state during the resolution will be burned by backtracking and the object gets back its initial state at the end of the resolution.

The preservation, during the resolution, of more than one state of the same object, is a characteristic of several hybrid languages integrating OOP and PL. But in these languages, only the last state is accessible. In LOP, all the memorised states in the object structure are accessible: the initial state is accessible from the object identifier, and the other ones are accessible from the identifiers generated by the interpreter. An object having such a behaviour is called a multiple states object.

The implementation of state changes is done in LOP by the predicate set. The reduction of a literal in the form obj< -set(F at1 NV1 ... atp NVp) produces a new state and an identifier that allows its access. The previous identifier is unified with the logical variable F. In the new state, the attributes ati which appear in the call of set have the new values NVi, and the non mentioned attributes preserve their values corresponding to the old state.

In the context of multiple states objects, the message sending must indicate the receiver state. The receiver id of a literal in the form id< -meth(...), can be either the object identifier, or a state identifier. In the first case, this corresponds to the object initial state. It is worth mentioning that it is only possible for the user to name the initial state identifier. Other states are accessible via the logical variables that appear as first arguments of set. For example, in the request:

?- tri1<-set(F ac 0.9), F<-isosceles(A).
  F = tri1.1, A = b.
  success.

the state represented by F in the message sending F< -isosceles(A), is that produced by the message tri1< -set(F ac 0.9). In this new state, the triangle is isosceles relatively to the vertex b (in the initial state the triangle was equilateral with length side equals to 1). In the same manner, one can apply the method set over any object state in order to produce a new one. In the request:

?- tri1<-set(F ac 0.9), F<-set(G ab 0.9),
  G<-isosceles(A).
  F = tri1.1, G = tri1.2, A = a.
  success.

the state represented by G has the values ab = 0.9, ac = 0.9 and bc = 1.

3.2 Contribution of multiple states in object-oriented programming

The explicit designation of the receiver state allows the interpretation of message sending (that can use state changes) regardless of the resolution strategy. More precisely, a request can produce the same result even if its literals are inverted. For example, the two requests:

?- tri1<-set(F ac 0.9), F<-equilateral(X).
  fail.

?- F<-equilateral(X), tri1<-set(F ac 0.9).
  fail.

produce the same result (in the second request, the message sending F< -equilateral() is delayed until the instantiation of the variable F). This is also the case of the two following requests:

?- tri1<-set(F ac 0.9), tri1<-equilateral(X).
  X = 1.
  success.

2The identifier tri1.1 is generated by LOP's interpreter.
This independence from the order of the resolution strategy constitutes the declarative contribution of multiple states object. Such a property cannot be satisfied in the languages where only the last state is accessible. For example, in Prolog++ [Mos94], it is possible to define the methods set and equilateral over the class triangle by the program:

```prolog
class triangle.
attribute ab, ac, bc.
set(ab V) :- ab := V.
set(ac V) :- ...
set(bc V) :- ...
equilateral(X) :- @ab=X, @ac=X, @bc=X.
```

If tri1 is an instance of the class triangle with the initial values: ab = ac = bc = 1, the two requests of the same literals:

?- tril<-set(ac 0.9), tril<-equilateral(X).
fail.

?- tril<-equilateral(X), tril<-set(ac 0.9).
x = 1.
success.
```

give different results. In Prolog++, the interpretation of the message sending tri1<-equilateral(X) depends on its location in the request. In this case, the programmer must take in account the resolution strategy. Thus, the declarative aspect is lost.

Another contribution of multiple states objects is the possibility to access to all the states of an object. This can be interesting if we want to compare two possible evolutions of an object from a given state. This is the case of the request:

?- tril<-set(F ac 0.9), F<-isosceles(A),
   tril<-set(G ab 0.9), G<-isosceles(B).
F = tril.1, A = b, G = tril.2, B = c.
success.
```

where the two new states F and G are produced from the initial state. This can be useful for the comparison and the choose of the best alternative for the resolution continuation.

### 3.3 The multiple definitions method

The definition of methods is usually accomplished ed in the languages which integrate OOP and PL via rules. It is possible in such languages to add rules, during the resolution, by assert and retract predicates (or similar ones). In this case, the initial definition is dynamically modified (i.e. redefined in the resolution phase). Such a dynamic redefinition goes against declarative programming. In fact, consider a request which contains a literal that modifies the definition of a method meth, as follows:

```prolog
... modif_def(meth[...]) :- ...,
... obj<-meth[...], ...
```

we suppose here that modif_def is a predicate that allows the redefinition of meth (using for example, assert and retract). In this case, the method meth used in obj<-meth(...) is that produced by modif_def(...). In the context of usual left-most literal resolution strategy. On the other hand, the request where the two literals are inverted:

```prolog
... obj<-meth[...],
... modif_def(meth[...]) :- ...
```

uses the initial definition of meth. As in the case of state changes, the message obj<-meth(...) is interpreted depending on the literals order.

The dynamic methods approach in LOP is similar to that of state changes. If a method is redefined during the resolution, its old and new definition remain accessible. The redefinition of a method over a class is considered as a state change of the class. The new definition is connected to the new state. Dynamic method can be create by the predefined method newdefm. A following message sending in the form:

```prolog
cla<-newdefm(F (rule1) ... (rulen));
```

where cla is the class identifier and rules are the rules which define the method, allows such a redefinition. The reduction of such a literal allows F to be unified to an identifier, generated by the interpreter, which allows the class new state to be consulted.

A state should be mentioned when a new definition is written. A literal in the form:

```prolog
id_cla@id_obj<-meth(...)
```

means that the receiver is considered in the state id_obj and the definition of the method meth is considered in the state id_cla of a given class. Note that if obj is an object identifier of a class cla, then the two messages obj<-meth(...) and cla@obj<-meth(...) are equivalent (the receiver and the method are both considered in their initial state and definition, respectively). In the two requests:

?- triangle_color<-newdefm(F (0<-reference() :-
   triangle@0<-reference(),
   0<-get(colour blue))),
   F@tri_coll<-reference().
fail.

?- F@tri_coll<-reference(),
   triangle_color<-newdefm(F (0<-reference() :-
   triangle@0<-reference(),
   0<-get(colour blue))).
F = triangle_color.l.
success.
```

the method reference() is considered according to its new definition. Thus the message sending F@tri_coll<-reference() fails because the triangle tri_coll is white. On the other hand, in the two requests:

?- triangle_color<-newdefm(F (0<-reference() :-
   triangle@0<-reference(),
   0<-get(colour blue))),
   tri_coll<-reference().
F = triangle_color.l.
success.
```

the method reference() is considered according to its new definition. Thus the message sending F@tri_coll<-reference() fails because the triangle tri_coll is white. On the other hand, in the two requests:

?- triangle_color<-newdefm(F (0<-reference() :-
   triangle@0<-reference(),
   0<-get(colour blue))),
   tri_coll<-reference().
F = triangle_color.l.
success.
```

the method reference() is considered according to its new definition. Thus the message sending F@tri_coll<-reference() fails because the triangle tri_coll is white. On the other hand, in the two requests:

?- triangle_color<-newdefm(F (0<-reference() :-
   triangle@0<-reference(),
   0<-get(colour blue))),
   tri_coll<-reference().
F = triangle_color.l.
success.
4 Implementation

LOP is implemented in CLOS under the form of hierarchical levels which correspond to different programming levels (NP1 to NP4). The basic level contains the ground of the object model of LOP and its scheme resolution. The second level NP2 inherits from NP1 and deals with multiple states object and dynamic methods. In NP3 some procedural mechanisms which allows the program control (such as cut, Prolog metalogic predicates: var, call, ...) are introduced. Finally, the last level enhances LOP by some types of constraints (linear constraints on naturals, ...) for discrete applications (simulation, ...).

4.1 The level NP1

The kernel of LOP's object model is written in CLOS as shown by the figure 4. The class metalop corresponds to the default metaclass of all LOP's user class. It allows the general structure of any class to be defined. This structure is composed from the attributes that collect the information related to the class (its super-classes, attributes, precedence list, ...). The class superlop is super-class of all the classes unless itself. Methods, such as get, ..., are (pre)defined over superlop.

For each program written in LOP, an internal representation is realised in CLOS. The translation from LOP to CLOS is done thanks to a pre-processor, implemented in CLOS. The translations realised by this pre-processor are the following:

- For each class defined by defclass(cl (supl ... supk) (atl ... atn)), a CLOS class is defined by:

  (defclass cl (supl ... supk)
   ((atl :initarg :atl) ...) {
    (atn :initarg :atn))
   (:metaclass metalop)).

- For each LOP's instance defined by definstance(cl obj (at1 v1) ... (atn vn)), a CLOS instance is defined by:

  (setq obj (make-instance cl :at1 v1 ... :atn vn))

- Each LOP's method defined by defmethod(cl (rule1) ... (rulen)), where rulei has the form: O< -meth(...):body, will be stored in the p-list of the symbol cl:

  (setf (get cl 'meth) (list rule1 ... rulen))

as well as the definition of the CLOS method meth of the form:

  (defmethod meth ((self cl)) (get cl 'meth)),

which allows the access to the rules which define meth, and ensure its inheritance.

The resolution scheme of LOP is an extension of that of Prolog to object literal reduction and freezing mechanism. A message in the form obj< -meth(...) will be reduced via each rule that defines meth over the class of obj.

The rules are obtained by the evaluation of the CLOS call: (meth obj) (using CLOS algorithm for precedence list order). On the other hand, a message in which the resolution class is specified: cl@obj< -meth(...) will be reduced by the rules that define meth over cl. These rules are accessible via the evaluation of CLOS call (meth obj'), where obj' is the object of the class cl created for this usage by (setq obj' (make-instance cl)).

The freezing mechanism is similar to that introduced in Prolog II.

4.2 The level NP2

As said before, the principal extension in NP2 is the management of multiple states object and dynamic methods. We describe here these two mechanisms, respectively.

The internal representation of a LOP object is a CLOS object, with the same state, which can be accessed by an object identifier. When a literal obj< -meth(...) is reduced, an identifier id is generated and its p-list is initialised to the list (object obj at NV). This last one contains two information: the object obj to which the new state, represented by id, is connected, and the new value NV of the modified attribute at. This structure is sufficient for the memorisation of all the information that is necessary for the remaining resolution process. For example, to reduce id< -meth(...), we should compute the class of the receiver. This is realised by the CLOS expression (class-of (get id 'objet)) which returns the class of the object related to id.
The reduction of the particular message \textit{id< }\textit{-get(at V)} needs another information. In this case, it is sufficient to consult the p-list associated with \textit{id}. If at appears in this p-list: (\textit{objet obj ... at NV ...}), then \textit{V} will be unified with the value returned by the expression: (slot-value at V). This expression means that the attribute value is taken from the initial state of the object \textit{obj}.

The last case is that of the message \textit{id< }\textit{-set(F at NV)}. Its reduction generates a new identifier \textit{id'} whose p-list is identical to that of \textit{id}, excepted what concerns the attribute \textit{at}. If this attribute appears in the p-list of \textit{at} : (\textit{objet obj ... at V ...}), then the p-list of \textit{id'} is (\textit{objet obj ... at NV ...}). On the other hand, if at does not appear in the p-list of \textit{id} then the p-list of \textit{id'} is that of \textit{id} in which the pair (at NV) is added.

Methods multiple definitions are managed similarly to the class dynamic creation of CLOS. The reduction of a literal in the form \textit{cl< }\textit{-newdefm(F (rule1) ... (rulen))}, where the rules \textit{rulei} (re)define a method \textit{meth}, generates an identifier \textit{idcl} which allows the new state of the class to be accessed.

In CLOS level, a new subclass \textit{idcl} of \textit{cl} plays the role of the new state of \textit{cl}. In fact, the unique difference between the state \textit{cl} of a given class and its new state \textit{idcl} is that the method \textit{meth} is defined differently in both states. This has the same behaviour if we consider, in CLOS level, that \textit{idcl} is the identifier of a subclass of \textit{cl} where the unique defined method is \textit{meth}. More precisely, a literal \textit{cl< }\textit{-newdefm(F (rule1) ... (rulen))} where the rules \textit{rulei} define a method \textit{meth} is reduced into the unification of \textit{F} with an identifier \textit{idcl} generated by CLOS, which satisfies:

\begin{itemize}
  \item[i-] (defclass \textit{idcl (cl)})
  \item[ii-] (setf (get \textit{idcl 'meth})
  \quad (list rule1 ... runen))
  \item[iii-] (defmethod \textit{meth} ((self \textit{cl}))
  \quad (get \textit{cl 'meth}))
  \item[iv-] (setf (get \textit{idcl 'class}) \textit{cl})
\end{itemize}

The subclass \textit{idcl} is created in the level i-. The new definition of \textit{meth} is memorised, in -ii, in the p-list of the symbol \textit{idcl}. The CLOS method which allows the access to the rules is defined in -iii. Observe that this is done only if \textit{meth} is not already defined over \textit{cl}. That is, the case -iii corresponds to the dynamic definition of a method and not to the redefinition of an already existing method. Finally, in -iv the identifier \textit{idcl} (which represents a state of the class \textit{cl}) memorises the class \textit{cl} in its p-list.

5 Comparison with related works

Many works related to the integration of LP and OOP have been recently presented. The complementary [27, 21] of these two paradigms encouraged several hybrid languages to be developed. The complete fusion between these two paradigms still an open problem. In fact, it is both necessary to find a logic that supports all OOP aspects, and to ensure an efficient implementation of a language based on this logic.

The problem of the state changes in hybrid languages illustrates the great difficulty to realise such an integration. Some languages exclude the dynamic aspects in order to preserve a pure logical framework. For instance, state changes are not authorised in Hayes [12], Zaniolo [27], POL [11] and LOGIN [4]. Objects in such languages are static [18]. Other languages such as Prolog++ [19], ESP [8] and ObjVProlog [17], authorise state changes and implement them via assert and retract like predicates as well as imperative assignment. Such predicates go against the declarative programming aspects [20, 15]. In particular, the semantic of a request depends here on the literals order. Because the classical logic is unsuitable for state change representation [5], some non classical logic have been proposed to support state changes of objects. This is the case of modal logic Mu [26]. The major problem of the use of this logic is that it is designed to reason about changes, but not to execute them [5]. More recently, the F-logic [14], proposed as a semantic base of OOP, was connected with TR logic [7] in order to support state changes of objects. We think that the complexity of their resolution model avoid their efficient use.

Another approach is that of LO [6], where linear logic is proposed as an alternative to classical logic, in order to support state changes of objects. These are realised similarly to CP's [25] streams of messages. These messages are collected in lists, and reduced according to their order in the lists. Every message is applied to the last state of the object.

In LOP, each message can be interpreted regardless of its position in the request and regardless of the resolution strategy. On the other hand, the user should specify the state of the receiver in which the message is sent. Moreover, our change state declarative approach is extended to dynamic methods redefinition, where this problem is solved using assert and retract like predicates in the previously mentioned hybrid languages.

Finally, LOP's resolution model has the same complexity as Prolog. In particular, there is no more logical connectors than those of classical logic. This is not the case of F-logic, TR or linear logic, where the complexity of the resolution model is increased.

References


Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1 ½ or double spacing.
- The first page should include:
  - the title (as brief as possible)
  - the author’s initials and surname
  - the author’s affiliation and address
  - an abstract of less than 200 words
  - an appropriate keyword list
  - a list of relevant Computing Review Categories
  - Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author’s surname, and should be cited in the text according to the Harvard method. References should also be according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj which gives a number of examples.

SACJ is produced using the \LaTeX\ document preparation system, in particular \LaTeX\ 2ε. Previous versions were produced using a style file for a much older version of \LaTeX, which is no longer supported. Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT. These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.
Contents

Editorial
P. Machanick .................................................. 1

Research Articles
Heuristics for Resolution-based Set-theoretic Proofs
J.A. van der Poll and W.A. Labuschagne ....................... 3

Orthogonal Ray Guarding of Adjacencies between Orthogonal Rectangles
I. Sanders, D. Lubinsky, M. Sears and D. Kourie ................ 18

Discriminators for Authorship Attribution
H. Paijmans ..................................................... 30

Electronic Performance Support Systems: Appropriate Technology for the Development of Middle Management in Developing Countries
J.C. Cronje and S.J. Baras Baker .................................. 42

A Formal Model for Objectbases
P.A. Patsouris, M. Korostenski and V. Kissimov .................. 54

Indexing in a Case-Based Reasoning System for Waste Management
K.L. Wortmann, D. Petkov and E. Senior ......................... 72

Technical Reports
Connected Digit Recognition in Afrikaans Using Hidden Markov Models
C. Nieuwoudt and E.C. Botha .................................... 85

A 3-Dimensional Security Classification for Information
W. Smuts .................................................................... 92

A declarative and non-determinist framework for Dynamic Object-Oriented and Constraint Logic Programming
H. Abdulrab, M. Ngomo and A. Drissi-Talbi ....................... 98

Communications and Viewpoints
Progressing towards Object Orientation in South Africa
M. Jansen van Rensburg ............................................. 107