International Journal of Computer Trends and Technology
ISSN: 2231 – 2803 / https://doi.org/10.14445/22312803/IJCTT-V69I10P102

Volume 69 Issue 10, 12-22, October 2021

Original Article

Structural Unification of the Logic Objects

Macaire Ngomo

CM IT CONSEIL – Engineering and Innovation Department – 32 rue Milford Haven 10100 Romilly sur Seine (France).

Received Date: 04 September 2021
Revised Date: 05 October 2021
Accepted Date: 17 October 2021

Abstract - *This paper is devoted to the structural unification of Logic Objects. The basic idea of the OO-Prolog language is the definition of a model in which objects are constructed by unification and undone by backtracking. In OO- Prolog, the resolution is based on a logical deduction of all consequences of the knowledge base via unification. In contrast, Prolog's unification mechanism is limited to syntactic processing and cannot consider the OO-Prolog language's objects. For the object layer of OO-Prolog to react homogeneously with the rest of the Prolog language, it is necessary to have a unification mechanism that considers Logic Objects. In this study, we propose building a unification scheme that considers the semantics of objects. This provides the user with a tool for comparing objects (from a structural point of view) and sharing data between objects. The algorithm we propose is limited to the case where the two objects to be unified belong to the same class. However, extensions of the algorithm to more general cases are proposed.*

Keywords - *Logic programming, Prolog, OO-Prolog, object programming, logic objects, unification, structural unification of logic objects.*

I. INTRODUCTION

Logic programming was developed in the early 1970s to extend earlier work on the automatic translation of theorems and artificial intelligence. Since logic sought to model human reasoning, it was hoped to simulate it on a computer. Building on the work done by Herbrand in 1930 [Herbrand 67], Prawitz [Prawitz 60], Gilmore [Gilmore 60a] [Gilmore 60b], Martin Davis and Hilary Putnam [Davis 60] and others, worked extensively on automatic theorem proving in the early 1960s. This effort culminated in 1965 with the central paper by Robinson [Robinson 65], which introduced the resolution rule. Solving is an inference rule that is particularly well suited to automation on a computer [Lloyd 87] [Lloyd 88].The credit for the introduction of logic programming goes mainly to Kowalski [Kowalski 70] [Kowalski 71] [Kowalski 74a] [Kowalski 74b] and Colmerauer [Colmerauer 72] [Colmerauer 73] [Colmerauer 75] [Colmerauer 92] [Colmerauer 93] even though others should be mentioned in this regard: Roussel [Roussel 72] [Roussel 75], Pasero Robert [Pasero 73], Jean Trudel [Colmerauer 71], Henry Kanoui [Kanoui 73], Battani [Battani 73], Green [Green 69], Hayes [Hayes 73], etc. In 72, Kowalski and Colmerauer came up with the fundamental idea that logic could be used as a programming language.The acronym Prolog (PROgramming in LOGic) was invented and the first Prolog interpreter [Colmerauer 73] [Colmerauer 75] was implemented in ALGOL-W by Roussel [Roussel 72] [Roussel 75] [Colmerauer 92] [Colmerauer 93], in Marseille (France) in 72.

Prolog (PROgramming in LOGic) was thus born out of the need to be able to process natural language by computer and, in particular, grammar [Colmerauer 73] [Colmerauer 75] [Colmerauer 82] [Colmerauer 89] [Colmerauer 92] [Colmerauer 93] [Clocksin-03][Cohen 88]. Since then, we can count many other application areas: Relational databases, Logic (and mathematics), Abstract problem solving, Natural language processing, Symbolic equation solving, Artificial intelligence, etc. Various implementations are available: Standard ISO, SWI-Prolog (Dept. of Social Science Informatics, www.swi-prolog.org), Prolog III (PrologIA), SICStusProlog (www.sics.se/sicstus), Open Prolog (www.cs.tcd. i.e./open-prolog), GNU Prolog, AllegroProlog, BProlog, Visual Prolog (Prolog Development Center A/S), YAP-Prolog, LPA-PROLOG, PoplogProlog, Turbo Prolog (Borland), IF-Prolog (Siemens Nixdorf), DelphiaProlog (Siglos), BIM Prolog (Integral Solutions Limited), Win-Prolog (Logic Programming Associates), PDC Prolog (Prolog Development Center), Quintus Prolog (AI International Limited), etc. (Comparison of Prolog implementations:
https://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations).

Hewitt's PLANNER system [Hewitt. 69] [Hewitt 70] [Hewitt 72] [Hewitt. 09] can be seen as a predecessor of Prolog. The idea that first-order logic or at least some of its substantial subsets could be used as a programming language was revolutionary at the time because, until 72, logic had only ever been used as a declarative or specification language in computer science [Lloyd 88]. The birth of logic programming meant the advent of specifications that computers could directly execute. However, Kowalski shows us in [Kowalski 74a] [Kowalski 74b] that logic has a procedural interpretation, which makes it very effective as a programming language.

One of the main ideas of logic programming, due to Kowalski [Kowalski 70] [Kowalski 71] [Kowalski 79a] [Kowalski 79b] [Kowalski 88], is that an algorithm is made up of two disjoint components, logic and control. The logic formulates the problem to be solved while the control formulates how to solve it [Kowalski 79a] [Kowalski 79b] [Kowalski 88] [Lloyd 87] [Lloyd 88]. In general, a logic programming system should provide the programmer with the means to specify these components. However, separating these two components has a number of advantages, not least of which is the ability for the programmer to specify only the logic component of an algorithm and leave control solely to the logic programming system itself [Lloyd 88]. In other words, an ideal logic programming system is purely declarative logic programming. Unfortunately, current logic programming systems have not yet achieved this.

In the object-oriented programming approach, there are several reasons to extend the syntactic unification of Prolog to the structural unification of objects.

A. Structural comparison of objects

Consider the following two objects :

id1 -> [class: car, name: 'R4', brand: 'Renault', year: 1978]

id2 -> [class: car, name: 'R4', brand: 'Renault', year: 1978]

For the system, there are two very distinct objects. On the other hand, their values are equal and we can say that these two objects are semantically unifiable (equal value in the terminology of object-oriented databases [Delobel 91]). In the following case :

id1 -> [class: car, name: 'R4', brand: 'Renault', year: 1978, chassis: id3]

id2 -> [class: car, name: 'R4', brand: 'Renault', year: 1978, chassis: id4]

whereid3 and id4 are object identifiers, with :

id3 -> [class: chassis, type: 'XF330', weight: 200]

id4 -> [class: chassis, type: 'XF330', weight: 200]

Although the identifiers id3 and id4 are not equal, they denote two semantically equal objects. Therefore, and by recursion, id1 and id2 are also semantically equal or "deep equal" in the terminology of some systems, for example, the Eiffel system. This equality means that the values may differ, but when we expand them by replacing the identifiers with the values of the objects they represent, we find two objects with the same structure. Everything happens as if we had :

id1 -> [class: car, name: 'R4', brand: 'Renault', year: 1978, chassis: id3]

id2 -> [class: car, name: 'R4', brand: 'Renault', year: 1978, chassis: id3].

The reader will notice in the latter case that the two cars would have shared the same chassis. This kind of distinction is easily expressed in object languages or object-oriented databases.

B. Information sharing

The above example brings us to the other important facet of object unification: data sharing between objects. If in the definition of id3 and id4 above, the "weight" field of id3 had the value of a free variable P and the "type" field of id4 had the value of a free variable T :

id3 -> [class: chassis, type: 'XF330', weight: P]

id4 -> [class: chassis, type: T, weight: 200]

then the unification of id3 and id4 or id1 and id2 allows to unify P with the constant 200 and the variable T with the value 'XF330'. Wethenobtain:

id3 -> [class: chassis, type: 'XF330', weight: 200]

id4 -> [class: chassis, type: 'XF330', weight: 200]

C. Formal unification

The core of the computational model of logic programs is the unification algorithm [Ait Kaci 91] [Boizumault 88] [Colmerauer 71] [Colmerauer 73] [Colmerauer 75] [Colmerauer 82] [Colmerauer 89] [Colmerauer 92] [Colmerauer 93] [Clocksin- 03] [Debarbieri 89] [Debarbieri 90] [Delahaye 86] [Delahaye 88] [Deransart 85] [Deransart 86] [Deransart 89a] [Deransart 89b] [Deransart 91] [Deransart 92] [Deransart 96] [Ferrand 92] [Kornfeld 83] [Kahn 81] [Jaffar 88] [ISO 93] [Geyres 89]. Unification allows us to determine, if it exists, the common instance of two terms. Unification is the basis of most of the work in automatic deduction and the use of logical inference in artificial intelligence [Debarbieri 89] [Debarbieri 90] [Delahaye 86] [Delahaye 88] [Ferrand 92] [Haton 91] [Huet 86] [Savory 06] [Shapiro 86] [Sethi 96] [Sterling 84] [Sterling 86a] [Sterling 86b] [Sterling 87] [Sterling 90] [Thaye 88] [Thaye 89] [Ueda 85] [Ueda 86] [Warren 77a] [Warren 77b] [Warren 83] [O'Keefe- 90] [ISO 93] [Geyres 89]. In Prolog, the resolution uses a parameter passing mechanism by substitution of variables between two terms [Boizumault 88] [Robinson 65]. The principle of unification is to find a substitution of a set of expressions such that all substituted expressions are identical. The concept of unification was introduced in 1930 by Herbrand [Herbrand 31]. It was discovered again in 1963 by Robinson to be exploited as a resolution filtering technique, which allowed to reduce the search space [Robinson 65]. We must therefore define what substitutions and unification are.

a) Substitution

Substitution is an application from the set of variables to the set of terms. It allows you to substitute a set of variables in formulas to obtain another formula.

A term t is a common instance of two terms t1 and t2 if substitutions □1 and □2 such that t equals □1t1 and □2t2. A term s is more general than a term t if t is an instance of s, but s is not an instance of t. A term s is an alphabetic variant of a term t if both s is an instance of t and t is an instance of s. A unifier of two terms is a substitution making the terms identical. If two terms have a unifier, we say that they unify. There is a close relationship between unifiers and common instances. Any unifier determines a common instance, and conversely, any common instance determines a unifier.

b) Most General Unifier » (mgu)

The question can be asked whether, for two given formulas, there is always a most general unifier. To do this, we must first define the differences between two formulas and then the algorithm that constructs the upg of two formulas. Unification is not a simple pattern matching that returns true if two terms are equal. Unification makes the two terms equal. And you need to understand this to program in Prolog.

A most general unifier or mgu of two terms is a unifier such that the associated common instance is the most general. If two terms unify, then there is a unique more general unifier. This uniqueness is subject to variable renaming. Equivalently, two unifiable terms have a unique most general common instance, at an alphabetic variant.

c) Unification algorithm

To fully understand the algorithm used by Prolog to run a program, one must first understand unification. Intuitively, unification attempts to determine whether there is a way to instantiate two expressions to make them equal. More formally, the unification of two terms, T1 and T2, is a substitution θ such that □T1 = □T2. We denote the unifier of T1 and T2. For example, the substitution {Y = joao, X = mother(joao)} is a unifier of the statements love(joao,X) and love(Y,mother_of(Y)).

Unification appears as a parameter passing mechanism but also as a tool for data selection and construction. The program ends when a resolution step produces a new empty goal.

If any, a unification algorithm calculates the more general unifier of two terms and displays failure otherwise. The algorithm for unification presented below is based on the solution of equations. The input to the algorithm consists of two terms T1 and T2. The algorithm's output is the mgu of the two terms if they unify or fail if they do not unify. The algorithm uses a stack to store the equations to be solved and a location □ to group the substitution that the output contains.

Input: two terms T1 and T2, to be unified
Output: □, the mgu of T1 and T2, or failure

Algorithm :

```
Initialize:
        the substitution □ to be empty, and the stack to contain
        the equation T1 = T2 and failure to false
While the stack is not empty and no failures occur, do:
        remove X = Y from the stack
        case:
                X is a variable with no occurrence in Y:
                        substitute Y for X in the stack
                        and in □ and add X = Y to □
                Y is a variable with no occurrence in X:
                        substitute X for Y in the stack
                        and in □ and add Y = X to □
                X and Y are identical constants or variables
                identical variables: continue X is f(X1,...,Xn) and Y is f(Y1,...,Yn)
                        for a functor f and n ≥ 1:
                        stack Xi = Yi, (i=1,...,n)
                        on the stack
                        else failure := true
End While

if failure, then output failure else output □
```

Intuitively, two predicate expressions T1 and T2, unify if they are formed from the same predicate, with the same number of arguments. Each argument of the predicate of T1 must unify with the argument at the same position in T2. Remember that each argument is itself a term, which can be a variable, a constant, or a compound term. Two terms unify if one of the following three conditions is met:

- both terms are identical constants ;
- one of the terms is an uninstantiated variable. In this case, the variable will be instantiated by the other term;
- the two terms have the same functor and the same number of arguments, and the arguments unify.

d) Algorithm for running a program

The unification algorithm, due to Robinson, allows to determine the most general unifier between two expressions [Boizumault 88] [Bratko 01] [Robinson 65] [Colmerauer 73] [Colmerauer 75] [Colmerauer 82] [Colmerauer 89] [Colmerauer 92] [Colmerauer 93]. In this algorithm, S is a finite set of simple expressions, and the mgu k represents the most general unifier of this set of formulas.

(1) k=0, αk =empty
(2) if Sαk is a singleton, the substitution αk is an mgu of S, else determine Dk the set of different Sαk.
(3) if there are v and t in Dk such that v is a variable not appearing in t, αk+1 = αk + {v ← t}, else S is not unifiable.

The vast majority of Prolog systems do not use the classical unification algorithm, making the deliberate choice not to perform the occurrence test (a variable can be unified to a term containing it) [Boizumault 88] [Bratko 01] [Colmerauer 73] [Colmerauer 75] [Colmerauer 82] [Colmerauer 89] [Colmerauer 92] [Colmerauer 93]. This choice is not without problems at the theoretical level since it challenges the Herbrand universe model limited to finite terms [Herbrand 30] [Herbrand 67]. At a more operational level, the implementation without special precautions of such an algorithm, ignoring the occurrence test, makes it prone to loops. Thus, unlike the original unification algorithm [Robinson 65] [Robinson 80a] [Robinson 80b], a variable may be linked to a term containing it. The main reason for this omission is a significant gain in execution time. Indeed, carrying out the occurrence test is a costly operation since for each creation of substitution {(x,t)}, it obliges to browse the whole term t to determine whether the variable x is in t or not.

A brief study of Robinson's unification algorithm shows us that it is difficult to achieve. Current implementations of the Prolog language use an algorithm based on the following:

Input: X and Y to be unified,
Output: one mguθ or unification failure.
Algorithm:
initialiseθ to empty,
drop on the stack (X, Y)
failure= false
while stack not empty and not failed do
unstack (X, Y)
depending on what
- X and Y are identical constants, continue
- X is a variable that does not appear in the term Y,
substitute X for Y in the stack
add to θ the substitution (X <- Y).
- Y is a variable that does not appear in the term X,
substitute Y for X in the stack
add to θ the substitution (Y <- X).
- X= f((Xi)n) and Y= f((Yi)n),
drop (Xi, Yi) in the stack, for all i
- default : failure = true
end (depending on what)
end (while).
if failure = true return unification failure
else return θ, the mgu of X and Y θ.

Thus, this unification algorithm finds a most general unifier (mgu) of two expressions when it exists. If it does not, it mentions it as a failure.

e) Resolution algorithm

One of the main ideas of logic programming due to Kowalski [Kowalski79a] [Kowalski79b] [Kowalski 88] is that an algorithm is made up of two components; the first component, the logic, makes it possible to specify a problem to be solved, the second component, the control, makes it possible to describe how the problem is to be solved. An ideal logic programming system would be to program in the purely declarative part, i.e., the logic component. Only the control part would be left to this system. Unfortunately, current logic programming systems have not yet achieved this, and the user needs to know how to express control at the clause level and how the system uses it to modify the solution of a problem. Essentially, a Prolog program is composed of rules, clauses declaring a fact that depends on other facts. These rules have the following form:

H:- B1,B2, . . .,Bn
H and B1, B2, . . ., Bn constitute the head and body of the rule, respectively.
The meaning of a rule is as follows: To solve a goal H, one must solve the conjunction of goals B1, B2, . . ., Bn. A fact can be considered as a clause without a body.
To satisfy a query, the Prolog interpreter uses the resolution algorithm, which uses a stack, called the resolver, to hold all the goals that remain to be solved (note that we consider here that a fact is a clause whose body is empty):
0) Position yourself in the first clause of the program.
1) Resolvent← query
2) While the resolvent is not empty :
2.1) Obj ← First objective of the resolvent
2.2) Remove the first objective from the resolvent
2.3) Find the first clause
C = H : -B1, B2, . . . Bn such that
H unifies with Obj
If we succeed
2.4) Add at the beginning of the resolvent all
goals B1,B2,...Bn
2.5) If there are other heads of clauses
unifiable with Obj
Memorize the position of the clause C
as the last point of choice
2.6) Return to 2.1
Else
2.7) If there are choice points:
Backtrack to the last point of choice
Return to step 2.3
Else :
Return FAIL
3) Return the instantiation of the variables of the query

It is important to note that in this process, the system stores the instantiated variables. When a variable is instantiated in a rule, either in the head or in the body, the same instantiation applies to all variable occurrences in the rule. When the interpreter backtracks to a choice point, it discards the result of all unifications that were performed between the time the choice point was stored and the time of failure.

f) Backtracking with several points of choice

When there is more than one choice point at the backtracking, care should be taken to identify which one will be considered first. Backtrack to the choice point corresponding to the last unified goal for which there were other unification possibilities.

II. UNIFICATION OF LOGIC OBJECTS: SIMPLE CASE

A. Principle

In the simple case, two instances i1 and i2, are semantically unifiable if the values of their respective attributes are unifiable in the Prolog sense. This definition does not take into account composite objects.

Consider two objects i1 = <a1 = v11, ..., an = vn1n> and i2 = <a1 = v21, ..., an = vn2n> two instances of a class C. The principle of the unification procedure in the simple case is "for any attribute aj of class C, unify the values v1j and v2j".

We then say that i1 and i2 are unifiable if the values of their common attributes are unifiable. Since the class of an object is represented by the value of the attribute class(#' Object'), this means that both objects must belong to the same class. For example if i1 = [class: chassis, type: 'XF330', weight: P] and i2 = [class: chassis, type: T, weight: 200]

P and T being free variables, then i1 and i2 are unifiable since T and 'XF330' are unifiable, likewise for P and 200. On the other hand, if i'1 = [class: chassis, type: 'XF630', weight: 200] then i1 and i'1 are not structurally unifiable since 'XF330' and 'XF630' are not unifiable. As in a psi-term, the arguments in this list of attributes of an object are indicated by the pairs (attribute, value) rather than by their position in the list. Therefore, the order of the attributes in this list is not important since these arguments are identified by the name of the attribute rather than by their position.

B. Algorithm

A formal description of the semantic unification algorithm in the simple case is as follows:

Input: two terms T1 and T2, to be unified
Output: □, the upg of T1 and T2, or failure
Algorithm:
Initialise: the substitution □ □to be empty, and
the stack to contain the equation
T1 = T2 and failure to false

Until stack is empty and no failures occur, do
unstack X = Y from the stack
case:
X is a variable with no occurrence in Y
occurrence in Y:
substitute Y for X in the stack and
in and add X = Y to □
Y is a variable with no occurrence in X
occurrence in X:
substitute X for Y in the stack and
in and add Y = X to □
X and Y are constants or variables identical:
continue
X is f(X1,...,Xn) and Y is f(Y1,...,Yn)
case:
f is # and n equals 1
case:
X1 is a variable that has no occurrence in Y1:
substitute Y1 for X1 in the stack and □ add X1 = Y1 to □
Y1 is a variable that has no occurrence in X1:
substitute X1 for Y1 in the stack and □ add Y1 = X1 to □
X1 and Y1 are atomic constants or identical variables: continue
X1 and Y1 are atomic constants that are not identical:
Calculate the sets E1 and E2 of the attributes of each of the objects X and Y.
if E1 and E2 are equal and E1 = E2 = {a1, ..., aj} then
Let V1 = [v11, ..., v1j] and V2 = [v21, ..., v2j] the respective lists of values of their respective their attributes.
stack V1 = V2 on the stack
else failure := true
else failure := true
else
for a functor f different from # and n ≥ 1: stack Xi = Yi, i=1,...,n on the stack
else failure := true
End Until
if failure, then output failure
then output□

The algorithm stops :

- in case of failure: L1 and L2 are not equal, or there is an attribute ai such that v1i and v2i (the values corresponding to the two objects) are not unifiable.
- on success: L1 and L2 are equal, and all attribute values are unifiable.

In OO-Prolog as in the object approach [Ngomo 1994a] [Ngomo 1994b] [Ngomo 1995a] [Ngomo 1995b] [Ngomo 1996] [Malenfant 90] [Meseguer 91] [Meseguer 92] [Meseguer 93] [McCabe 92], an object has a unique identifier, which makes it possible to distinguish objects. Two distinct objects cannot have the same identifier. In this case, applying the Prolog unification procedure to two OO-Prolog objects will always fail since two distinct identifiers can never be unified. Therefore, we need to modify the classic Prolog unification procedure to take into account the objects and the inheritance relationship.

For the object layer to react homogeneously with the rest of the Prolog language, it must have mechanisms identical to those of all data types present in Prolog. We propose to define a specific mechanism to take objects into account. In the case of objects, the use of this primitive poses the problem of names referencing objects. Two structurally unifiable objects can have different identifiers. There is no valid justification for accepting a success for the unification of different object names while the unification of distinct functor terms fails even if all other elements composing the terms are identical [Cervoni 94]. As a consequence, we are obliged to have specific operators for object names. In OO-Prolog, the name of an object is preceded by the operator #:# <name>.

For example, "#' Point'" instead of 'Point'. This notation is used to distinguish object names from other Prolog terms.

C. Example

Declaration of the Car class:

```
#' Car' with
[ class(#'Object') := #'Class',
inherits(#'Class') := [#'Object'],
attributes(#'Class') := [
                    name(#'Car'),
                    brand(#'Car'),
                    year(#'Car') <=
                            year:number(Year),
                    chassis(#'Car') <=
                            Chassis:#'Chassis']
].
```

Declaration of the Chassis class:

```
#' Chassis' with
[       class(#'Object') := #'Class',
        inherits(#'Class') := [#'Object'],
        attributes(#'Class') :=
                [type(#'Chassis'),weight(#'Chassi
                s') <= P:number(P)]
].
```

Example of structural comparison of objects

```
> #'Chassis' <- new(C1,[type(_) := 'XF330',poids(_)
:= 200]),
#'Chassis' <- new(C2,[type(_) := 'XF330', weight(_)
:= 200]),
C1 <- unify(C2),
(C1,C2) <- display.

TERMINAL :: #[#'Chassis',1]
type(#'Chassis') <- 'XF330'
weight(#'Chassis') <- 200
class(#'Object') <- #'Chassis'

TERMINAL :: #[#'Chassis',2]
type(#'Chassis') <- 'XF330'
weight(#'Chassis') <- 200
class(#'Object') <-  #'Chassis'

{C1 = #[#'Chassis',1],C2 = #[#'Chassis',2]}
true
```

Example of data sharing

```
> #'Chassis' <- new(C1,[type(_) := 'XF330',poids(_)
:= P]),
#'Chassis' <- new(C2,[type(_) := T, weight(_) :=
200]),
C1 <- unify(C2),(C1,C2) <- display.

TERMINAL :: #[#'Chassis',1]
type(#'Chassis') <- 'XF330'
weight(#'Chassis') <- 200
class(#'Object') <- #'Chassis'

TERMINAL :: #[#'Chassis',2]
type(#Chassis) <- 'XF330'
weight(#Chassis) <- 200
class(#'Object') <-  #'Chassis'

{C1 = #[#'Chassis',1],P = 200,
C2 = #[#'Chassis',2], T = 'XF330'}
```

III. RECURSIVE UNIFICATION

The algorithm described above does not take into account composite objects. Indeed, suppose the values to be unified are names of distinct but semantically unifiable objects. In that case, the unification procedure will fail since the two objects have non-unifiable identifiers in the sense of Prolog. To take into account composite objects, the above

definition will be improved as follows: two instances i1 and i2, are semantically unifiable if and only if the values of their respective attributes are unifiable in the Prolog sense or semantically unifiable.

A. Principle

In the case of composite objects, the procedure is a little more complex. Indeed, if the values of the attributes are objects, one cannot proceed as in the case of simple objects, i.e., unify their identifiers. Since two non-equal objects cannot have the same identifier, the unification procedure would fail. Recall that we want to unify objects structurally, not syntactically. Of course, syntactic unification automatically leads to semantic unification. To unify composite objects semantically, they must be flattened, i.e., when the values of the attributes are objects, the semantic unification procedure must be applied by recursion to these two objects while managing any cycles. In this case, the objects

id1 -> [class: car, name: 'R4', make: 'Renault', year: 1978, chassis: id3]

id2 -> [class: car, name: 'R4', make: 'Renault', year: 1978, chassis: id4]

will be semantically unifiable, the other values being unifiable if id3 and id4 are also unifiable. We must then enrich the above algorithm by including the case of object values.

B. Algorithm

The algorithm we describe here is limited to the case where the two objects to be unified belong to the same class. We will see in the following how to extend this algorithm to other situations. Currently, the algorithm is limited to the case where the two objects to be unified belong to the same class. For two objects with different classes, the classification mechanism can assign them to the same class. In the algorithm below, we assume that the two objects belong to the same class.

Input : wo terms T1 and T2 to be unified

Output : □, mgu of T1 and T2, or failure

Algorithm:

Initialise:

the substitution to be empty,

the stack to contain the equation T1 = T2,

and failure is false

Until the stack is empty and there is no failure do

remove X = Y from the stack

case:

X is a variable that has no occurrence in Y:

substitute Y for X in the stack and in and add X = Y to □

Y is a variable that has no occurrence in X:

substitute X for Y in the stack and in and add Y = X to □

X and Y are identical constants or variables: continue

X is f(X1,...,Xn) and Y is f(Y1,...,Yn)

case:

f is # andn is 1

case:

X1 is a variable with no occurrence in Y1:

substitute Y1 for X1

in the stack

and in□ □add X1 = Y1 to□

Y1 is a variable with no occurrence in X1:

substitute X1 for Y1

in the stack

and in □□add Y1 = X1 to □

X1 and Y1 are atomic constants or identical variables: continue

X1 and Y1 are atomic constants that are not identical:

Calculate the sets E1 and E2 of the attributes of objects X and Y.

If L1 and L2 are

equal (L1 = L2 =
{a1, ...,aj})
Let V1 = [V11, ...,
V1j] and V2 = [V21, ..., V2j]
stack V1i = V2i, i=1,...,n on the stack
elsefailure := true
else failure := true
else for a functor f different from # and n ≥ 1:
add Xi = Yi, i=1,...,n on the stack
else failure := true
End Until
If failure, thenoutput*failure*
elseoutput□

Let us emphasize here the role of the object identifier. It offers flexibility in manipulating objects, and an object identifier also allows information to be shared without duplication. Indeed, since an object can reference another object directly by its identifier, the direct consequence is that the referencing object shares all the information contained in the referenced object, and any modification of the referenced object is directly visible if accessed from the referencing object. Similar to what can happen with Prolog terms, unification is defeated on a rollback. An implementation of this algorithm in Prolog is provided in [Ngomo 1996].

C. Example

Example of a structural comparison:

```
> #'Car' <- new(V1,[
      name(#'Car') := 'R4',
      brand(#'Car') := 'Renault',
      year(#'Car') := 1978,
      chassis(#'Car') := C1]),
  #'Car' <- new(V2,[
      name(#'Car') := 'R4',
      brand(#'Car') := 'Renault',
      year(#'Car') := 1978,
      chassis(#'Car') := C2]),
  #'Chassis' <- new(C1,[type(#'Chassis') :=
       'XF330',weight(#'Chassis') := 200]),
  #'Chassis' <- new(C2,[type(#'Chassis') :=
       'XF330',weight(#'Chassis') := 200]),
      V1 <- unify(V2), (V1,V2) <- display.

  TERMINAL :: #[#'Car',1]
  name(#'Car') <- 'R4
  brand(#'Car') <- 'Renault'
  year(#'Car') <- 1978
  chassis(#'Car') <- #[#'Chassis',3]
  class(#'Object') <- #'Car'

  TERMINAL :: #[#'Car',2]
  name(#'Car') <- 'R4
  brand(#'Car') <- 'Renault'
  year(#'Car') <- 1978
  chassis(#'Car') <- #[#'Chassis',4]
  class(#'Object') <- #'Car'

  {V1 = #[#'Car',1],C1 = #[#'Chassis',3],V2 = #[#'Car',2],C2 = #[#'Chassis',4]}
```

Example of data sharing between objects:

```
> #'Chassis' <- new(C1,[type(#'Chassis') :=
  'XF330',weight(#'Chassis') := P]),
  #'Chassis' <- new(C2,[type(#'Chassis') :=
   T,weight(#'Chassis') := 200]),C1 <- unify(C2),
  all(X,member(X,[C1,C2]))  <- display.

      TERMINAL :: #[#'Chassis',1]
            type(#'Chassis') <- 'XF330'
            weight(#'Chassis') <- 200
            class(#'Object') <- #'Chassis'

      TERMINAL :: #[#'Chassis',2]
            type(#'Chassis') <- 'XF330'
            weight(#'Chassis') <- 200
            class(#'Object') <- #'Chassis'
      {C1 = #[#'Chassis',1],C2 = #[#'Chassis',2]}
  > #'Car' <- new(V1,[
```

```
name(#'Car') := 'R4',
brand(#'Car') := 'Renault',
year(#'Car') := 1978,
chassis(#'Car') := C1]),
#'Car' := 'Renault <- new(V2,[
      name(#'Car') := 'R4',
      brand(#'Car') := 'Renault',
      year(#'Car') := 1978,
      chassis(#'Car') := C2]),
#'Chassis' <- new(C1,[type(#'Chassis') :=
      'XF330',weight(#'Chassis') := P]),
#'Chassis' <- new(C2,[type(#'Chassis') :=
      T,weight(#'Chassis') := 200]),
V1 <- unify(V2), (V1,V2) <- display.

TERMINAL :: #[#'Car',1]
name(#'Car') <- R4
brand(#'Car') <- Renault
year(#'Car') <- 1978
chassis(#'Car') <- #[#'Chassis',3]
class(#'Object') <- #'Car'

TERMINAL :: #[#'Car',2]
name(#'Car') <- R4
make(#'Car') <- 'Renault'
year(#'Car') <- 1978
chassis(#'Car') <- #[#'Chassis',4]
class(#'Object') <- #'Car'

{V1 = #[#'Car',1],C1 = #[#'Chassis',3],V2 = #[#'Car',2],C2
= #[#'Chassis',4],P=200,T='XF330'}
```

IV. SOME EXTENSIONS OF THE STRUCTURAL UNIFICATION ALGORITHM FOR LOGIC OBJECTS

The unification of two terms of the same class consists mainly in recursively unifying the fields of the structures of the instances. In the case of objects, two instances of the same class are semantically unifiable if and only if the values of their respective attributes are unifiable in the Prolog sense or semantically unifiable. This definition does not allow us, for example, to unify a rectangle of length 4 and width 4 with a square of side 4 unless we use the classification mechanism. Again, the user must explicitly express the classification constraints. If the instances are not of the same class, we must look for a possible inheritance relationship between them that would allow us to unify them by specialization or generalization. We propose below some extensions of the above algorithm. These extensions are not yet implemented in OO-Prolog but are verified.

V. CONCLUSION

In this paper, we have presented an algorithm for extending the syntactic unification of Prolog to the structural unification of logic objects. The algorithm currently implemented in OO-Prolog is limited to the case where the two objects to be unified belong to the same class. The extensions proposed in this study remain to be implemented.

ACKNOWLEDGMENT

The author wishes to thank Habib Abdulrab, Jean-Pierre Pécuchet, AbdenbiDrissi-Talbi, Mohamed Rezrazi, Fabrice Sebbe, and all his friends and colleagues for their help and support. He also wishes to thank Olga, Michel, Marielle, and Guyriel, who has always been very precious support for the realization of this work.

REFERENCES

[1] [Ait Kaci 91] Ait Kaci H., Warren's abstract machine: a tutorial reconstruction, MIT Press, 1991.

[2] [Battani 73] G. Battani and H. Meloni, Interpreteur du langage de programmation Prolog, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, (1973).

[3] [Boizumault 88] Boizumault, P. Prolog l'implantation, Masson (1988), 303.

[4] [Bratko01] Ivan Bratko: Prolog Programming for the Artificial Intelligence. 3rd Ed., Addison-Wesley, Harlow (UK), (2001).

[5] [Cervoni 94] L. Cervoni "Méthodologies et Techniques de résolution de Problèmes avec Contraintes. Application en Programmation Logique avec Objets : CooXi.Thèse de Doctorat Nouveau Régime, Université de Rouen, juillet (1994).

[6] [Clocksin-03] William F. Clocksin, Christopher S. Mellish : Programming in Prolog. 5th Ed., Springer-Verlag, Berlin, (2003).

[7] [Cohen 88] Cohen J., A View of the Origins and Development of Prolog, Communications of the ACM, 31(1) (1988) 26-36.

[8] [Colmerauer 71] Colmerauer Alain, Fernand Didier, Robert Pasero, Philippe Roussel, Jean Trudel, Répondre à, publication interne, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, mai 1971. Cette publication consiste en un listing avec des commentaires à la main.

[9] [Colmerauer 72] Colmerauer Alain, Henry Kanoui, Robert Pasero et Philippe Roussel, Un système de communication en français, rapport préliminaire de fin de contrat IRIA, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, octobre (1972).

[10] [Colmerauer 73] A. Colmerauer, P. Roussel and R. Pasero, Un système de communication Homme-Machine en Français, Groupe d'Intelligence Artificielle, Univ. d'Aix-Marseille, (1973).

[11] [Colmerauer75] Colmerauer Alain, Les grammaires de métamorphose GIA, publication interne, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, novembre 1975. Version anglaise, Metamorphosisgrammars, Natural Language Communication with Computers, Lectures Notes in Computer Science 63, édité par L. Bolc, Springer Verlag, Berlin Heidelberg, New York, pages 133 à 189, 1978, ISBN 3-540-08911-X.

[12] [Colmerauer 82] A. Colmerauer Prolog II : Manuel de Référence et Modèle Théorique, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, France, (1982).

[13] [Colmerauer 89] A. Colmerauer. Une introduction à Prolog III. Journée de Synthèse AFECT, Etat de 1' Art et Perspectives en Programmation Language, INRIA, (1989) 129-155.

[14] [Colmerauer 92] Alain Colmerauer et Philippe Roussel: La naissance de Prolog, juillet (1992).

[15] [Colmerauer 93] A. Colmerauer, The Birth of Prolog. In The Second ACM-SIGPLAN History of Programming Languages Conference pages 37-52. ACM SIGPLAN Notices, March (1993).

[16] [Davis 60] Martin Davis and Hilary Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM, 7(3) (1960) 201–215 (DOI 10.1145/321033.321034).

[17] [Debarbieri89] Christian Debarbieri. Mise au point en Prolog, Une vérification de la résolution. Congrès AFCET-RFIA'89, Paris 1989.

[18] [Debarbieri90] Christian Debarbieri: Étude et Réalisation d'un Système d'Aide à la Mise au Point en Programmation Logique. Thèse de doctorat de l'Université de Saint-Étienne et de l'École nationale supérieure des mines de Saint-Étienne, France, (1990).

[19] [Delahaye 86] J.P. Delahaye. Outils logiques pour l'intelligence artificielle. Editions Eyrolles. (1986).

[20] [Delahaye 88] J.-P. Delahaye, Outils logiques pour l'intelligence artificielle, Eyrolles, (1988).

[21] [Delobel 91] Claude Delobel, Michael Kifer, et al..Deductive and Object-oriented Databases: Proceedings of The Second International Conference, Dood'91, Munich, Germany, December 16-18, (1991).

[22] [Deransart 85] P. Deransart, G.Richard, C. Moss. Spécifications formelles de Prolog Standard. Programmation Logique, actes du séminaire (1985), CNET.

[23] [Deransart 86] P. Deransart, "Quelques idées pour une spéfication de la sémantique de Prolog". Rapport de Recherche INRIA n° xx, Février (1986).

[24] [Deransart 87] Pierre Deransart& Gérard Ferrand, An OperationalFormalDefinition of Prolog, Rapport de Recherche INRIA-Roquencourtn° 763, Décembre (1987).

[25] [Deransart89a] Pierre Deransart& Gérard Ferrand. A methodological view of logic programming with negation. Rapport de Recherche No 1011, INRIA-Roquencourt, Avril1989.

[26] [Deransart 89b] Pierre Deransart& Gérard Ferrand. Proofs Methods and Declarative Diagnosis in Logic Programming. ICLP'89, June 19-23 Lisbonne.

[27] [Deransart 91] P. Deransart, Ferrand G., A Formal Operational Definition of Prolog: A Specification Method and Its Application, Rapport de recherche INRIA, revised manuscript, (1991).

[28] [Deransart 92] P. Deransart, G. Ferrand, A Formal Operational Definition of Prolog: A Specification Method and its Application, New Generation Computing, 10 (1992) 121-171.

[29] [Deransart 96] Deransart, Pierre, Ed-Dbali, AbdelAli, Cervoni, Laurent, Prolog: The Standard: Reference Manual, Springer Verlag, Avril (1996), ISBN: 3-540-59304-7.

[30] [Ferrand 92] G. Ferrand, P. Deransart, Prolog method of partial correctness and weak completeness for normal logic programs, MIT Press, JICSLP'92, Washington, Nov. 9-13 1992.

[31] [Geyres 89] Stéphane Geyres. Une approche industrielle de la validation et de la vérification des systèmes à base de connaissances. Génie Logiciel et Système Expert, n 16. Ed EC2, septembre (1989).

[32] [Gilmore 60a] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. IBM Journal of research and development, 4(1960) 28–35.

[33] [Gilmore 60b] P. C. Gilmore. A program for the production from axioms of proofs for theorems derivable within the first-order predicate calculus. English, with English, French, German, Russian, and Spanish summaries. Information processing, Proceedings of the International Conference on Information Processing, Unesco, Paris 15–20 June 1959, Unesco, Paris, R. Oldenbourg, Munich, Butterworths, London, (1960) 265–273.

[34] [Green 69] C. Green. Theorem-proving by resolution as a basis for questions-answering systems. In B. Meltzer and D. Michie, editors, Machine Intelligence 4, pages 183-205, Edinburgh University Press, Edinburgh, (1969).

[35] [Haton 91] Haton, P. Le Raisonnement en Intelligence Artificielle. InterEditions, (1990).

[36] [Hayes 73] P. J. Computation and Deduction, Proc. MFCS Conf., Czechoslovak Academy of Sciences, (1973) 105-118.

[37] [Herbrand31] J. Herbrand. Sur le problème fondamental de la logique mathématique. Comptes rendues des séances de la société des sciences et des lettres de varsovie. Cl III, 24 (1931).

[38] [Herbrand 67] J. Herbrand, Investigations in Proof Theory, in From Frege to Gödel : A Source Book in Mathematical Logic, 1879-1931, van Heijenoort, J. (ed.), Harvard University Press, Cambridge, Mass. (1967) 525-581.

[39] [Hewitt. 69] Carl Hewitt. PLANNER: A Language for Proving Theorems in Robots. IJCAI 1969: 295-302

[40] [Hewitt. 70] Carl Hewitt. PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, Massachusetts Institute of Technology, Project MAC, Artificial Intelligence, Memo No 168, 1970-08-01.

[41] [Hewitt. 72] Carl Hewitt. Description and Theoretical Analysis (Using Schemata) of Planner, A Language for Proving Theorems and Manipulating Models in a Robot AI Memo No. 251, MIT Project MAC, (1972).

[42] [Hewitt. 09] Carl Hewitt. Middle History of Logic Programming: Resolution, Planner, Prolog and the Japanese Fifth Generation Project ArXiv 2009.

[43] [Huet 86] G. Huet, Deduction and Computation, Rapport de Recherche INRIA, N° 513, Avril (1986).

[44] [ISO 93] ISO (1993). Draft Standard for the Programming Language Prolog, ISO/IEC CD 13211-1: 1993 (E).

[45] [Jaffar 88] J. Jaffar and J.-L. Lassez. From unification to constraints. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, Logic Programming '87: Proc. 6th Conf., Tokyo, Japan, (1987). Springer-Verlag, Berlin, 1988. (LNCS, 315).

[46] [Kahn 81] K. M. Kahn. UNIFORM: a language-based upon unification which unifies (much of) LISP, PROLOG, and ACT 1. In IJCAI'81, (1981) 933-939.

[47] [Kanoui, 1973] Kanoui Henry, Application de la démonstration automatique aux manipulations algébriques et à l'intégration formelle sur ordinateur, thèse de 3ème cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, octobre (1973).

[48] [Kornfeld 83] Kornfeld W. A., Equality for Prolog, Proc. of IJCAI'83, (1983) 514-519.

[49] [Kowalski 70] Kowalski, R. A. Kowalski and D. Kuchener, Resolution with selection function, Artificial Intelligence, 2 (3) (1970) 227-260.

[50] [Kowalski 71] Kowalski Robert A. et D. Kuehner, Linear resolution with selection function, memo 78, University of Edinburgh, School of Artificial Intelligence, 1971. Aussidans Artificial Intelligence 2, 3, pages 227 à 260.

[51] [Kowalski 74a] Kowalski, R. Predicate Logic as a Programming Language, Information Processing 74, Stockholm, North-Holland, 1974, 1974, 569-574.

[52] [Kowalski 74b] Kowalski Robert A. and Maarten van Emden, The semantic of predicate logic as the programming language, memo 78, University of Edinburgh, School of Artificial Intelligence, 1974. Aussi dans JACM 22, 1976, pages 733 à 742. (sémantique moderne par point fixe de la programmation avec clauses de Horn)

[53] [Kowalski 79a] Kowalski, R. Algorithm = Logic + Control, Comm. ACM 22, 7 (1979), 424-436.

[54] [Kowalski 79b] Kowalski, R. Logic for problem solving. North-Holland, Amsterdam, (1979).

[55] [Kowalski 88] Kowalski Robert A., The early history of logic programming, CACM, 31(1) (1988) 38-43.

[56] [Lloyd 88] [Lloyd 87] John W. Lloyd: Foundations of Logic Programming, 2nd Edition. Springer (1987), ISBN 3-540-18199-7.

[57] [Lloyd 88] John W. Lloyd: Directions for Meta-Programming. FGCS (1988) 609-617.

[58] [Malenfant 90] Malenfant, J. Conception et Implantation d'un langage de programmation intégrant trois paradigmes: la programmation logique, la programmation par objets et la programmation répartie. Thèse de PhD, Univ. de Montréal, Mars (1990).

[59] [McCabe 92] F. G. McCabe. Logic & Objects. International Series in Computer Science. Prentice-Hall, (1992).

[60] [Meseguer 91] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96(1):73 155, 1992. Also Technical Report SRI CSL 91 05, SRI International, Feb. 1991.

[61] [Meseguer 92] J. Meseguer. Multiparadigm logic programming. In Third Intl. Conf. on Algebraic and Logic Programming, pages 158 200, Volterra, Italy, Sept. (1992).

[62] [Meseguer 93] J. Meseguer and X. Qian. A logicd semantics for object-oriented databases. In A CM SIGMOD Conference on Management of Data, (1993) 89 98, New Yok}, ACM.

[63] [Ngomo 94a] Ngomo M. & Pécuchet J-P. Contribution à l'étude de l'association des paradigmes de programmation logique et programmation par objets. Poster RJC-IA'94, (1994) 314 Marseille, France.

[64] [Ngomo 94b] Ngomo M. & Pécuchet J-P. Contribution à l'étude de l'association des paradigmes de programmation logique et programmation par objets. Actes du 2ème Colloque Africain sur la Recherche en Informatique, CARI'94, (1994) 879-893, Ouagadougou, Burkina Faso.

[65] [Ngomo 95a] Ngomo M. , Pécuchet J-P. & Drissi-Talbi A. Une approche déclarative et non-déterministe de la programmation logique par objets mutables. Actes des 4èmes Journées Francophones de Programmation Logique et Journées d'étude Programmation par contraintes et applications industrielles, Prototype JFPLC'95, (1995) 391-396.

[66] [Ngomo 95c] Ngomo M., Pécuchet J-P., Drissi-Talbi A. Intégration des paradigmes de programmation logique et de programmation par objets : une approche déclarative et non-déterministe. Actes du 2ème Congrès bienal de l'Association Française des Sciences et Technologies de l'Information et des Systèmes, AFCET - Technologie Objet – 95, (1995) 85-94, France.

[67] [Ngomo 96] Ngomo M. Intégration de la programmation logique et de la programmation par objets : étude, conception et implantation. Thèse de Doctorat d'Informatique, Université de Rouen - INSA de Rouen, Décembre (1996).

[68] [O'Keefe-90] Richard A. O'Keefe: The Craft of Prolog. The MIT Press, Cambridge (MA), (1990).

[69] [Pasero 73] PaseroRobert , Représentation du français en logique du premier ordre en vue de dialoguer avec un ordinateur, thèse de 3ème cycle, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, mai (1973).

[70] [Prawitz 60] Dag Prawitz, An improved proof procedure., First published: August 26(2) (1960) 102-139. https://doi.org/10.1111/j.1755-2567.1960.tb00558.

[71] [Robinson 65] Robinson, J.A. A Machine-Oriented Logic Based On the Resolution Principle. J. ACM 12, 1, (1965) 23-41. (sur le principe de résolution).

[72] [Robinson 80a] J.A. Robinson, E.E. Sibert, Loglisp an alternative to Prolog, Research Report 80-7, University of Syracuse, (1980).

[73] [Robinson 80b] J.A. Robinson, E.E. Sibert, Logic Programming in Lisp, Research Report 80-8, University of Syracuse, (1980).

[74] [Roussel 72] Roussel Philippe, Définition et traitement de l'égalité formelle en démonstration automatique, thèse de 3ième cycle, Groupe Intelligence Artificielle, Faculté des SciencesdeLuminy, Université Aix-Marseille II, France, mai (1972).

[75] [Roussel 75] Prolog, manuel de référence et d'utilisation, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, septembre (1975).

[76] [Savory 06] Jacques Savoy : Introduction à la programmation logique (Prolog), (2006).

[77] [Sethi 96] Ravi Sethi : Programming Languages : Concepts & Constructs. Addison-Wesley, Reading (MA), 1996. (G6-1554).

[78] [Shapiro 86] L. Sterling andE. Shapiro. The Art of Prolog. Advanced programming techniques. The MIT Press, Cambridge, Massachusetts, (1986).

[79] [Sterling 84] L. Sterling. Expert System= Knowledge + Meta-Interpreters. Technical Report No CS84-17. Weizmann institute of Science, 7 6100 Rehovot, Israel.

[80] [Sterling 86a] Leon Sterling. Incremental Flavor-mixing of Meta-interpreters for Expert system construction. Proceedings of Symposium on logic programming, Salk lake city, Utah, 20-27, 1986.

[81] [Sterling 86b] Leon Sterling, Ehud Shapiro: The Art of Prolog: Advanced Programming Techniques. The MIT Press, Cambridge (MA), (1986). (G6-139).

[82] [Sterling 87] L. Sterling & A. Lakhotia. Composing Prolog Meta-interpreters. Case Western Reserve University, Cleveland Ohio 44106 USA. (1987).

[83] [Sterling 90] Sterling, L. et Shapiro, E. L'Art de Prolog. MASSON (1990).

[84] [Thaye 88] Thaye, A. &co. Approches Logiques de l'Intelligence Artificielle: de la logique classique à la programmation logique. Dunod Informatique, (1988).

[85] [Thaye 89] Thaye, A. &co. Approches Logiques de l'Intelligence Artificielle: de la logique modale à la logique des bases de données. DunodInformatique, (1989).

[86] [Ueda 85] Ueda, K. Guarded Horn Clauses, Actes Logic Programming'85 LNCS 221 (1985) 168 179.

[87] [Ueda 86] K. Ueda, Guarded Horn Clauses, Ph.D. Thesis, University of Tokyo, (1986).

[88] [Warren 77a] D.R. Warren. Implementing Prolog: compiling predicates logic programs. D.A.I research port n 39/40, university of Edinbough, (1977).

[89] [Warren 77b] Warren David H. D., Luis M. Pereira et Fernando Pereira, Prolog the language and its implementation, Proceedings of the ACM, Symposium on Artificial Intelligence and Programming Languages, Rochester, N.Y., aout (1977).

[90] [Warren 83] D.R. Warren. An abstract Prolog instruction set. Technical note 309, SRI international, Menlo Park, (Octobre 1983), 30.