Semantics of the concurrent logic programming language PARLOG

G. Richard, A. Rizk

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 848

SEMANTICS OF THE CONCURRENT LOGIC PROGRAMMING LANGUAGE PARLOG

Gilles RICHARD
Antoine RIZK

MAI 1988

Semantics of the Concurrent Logic Programming Language PARLOG

Gilles RICHARD - Antoine RIZK
INRIA
Domaine de Voluceau
B.P.105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36

Abstract:

Horn clauses with negation, due to their clear and simple semantics [DF 87a], have recently proved to be a very powerful and flexible language for the construction of a formal specification for standard PROLOG (documents BSI/AFNOR PS 198-PS210) [DR87] .
During the last few years, the idea of coupling logic programming with concurrent programming has given rise to new programming languages : Concurrent Prolog [Sha 86], Guarded Horn Clauses [Ued 85b] and PARLOG [CG 86].
Although based on Horn clauses, these languages differ from PROLOG in two essential properties :
- they are non-deterministic
- their interpretation makes use of **and** and **or** parallelism and interprocess communication and synchronisation is done through specialised unification.

The semantics of these languages, however, remains little explored to date ([SAR 86], [Beck 86], [Ued 85a]).
The development of a PARLOG compiler in our project [GJR 87] (in the framework of ESPRIT) has incited us to write a formal specification of the operational semantics of this language using the methods already established for Standard PROLOG.
The realisation of this specification allows us to conclude that it is entirely possible to describe the inherent nondeterminism of these languages, and to widen the scope of application for this method of specification.

Keywords:

Formal specification, operational semantics, non-determinism, parallelism, concurrent logic programming, PARLOG semantics.

Note : this work has been partially supported by the Ministère de la Recherche et de l'Enseignement Supérieur, the GRECO-METHEOL and the ESPRIT project#956 COCOS.

Sémantique de PARLOG, un langage logique parallèle

Gilles RICHARD - Antoine RIZK
INRIA
Domaine de Voluceau
B.P.105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36

Résumé :

L'utilisation des clauses de Horn avec négation comme langage de spécification a récemment servi pour la réalisation de la norme PROLOG (documents BSI/AFNOR PS/198-PS/210). Doté d'une sémantique déclarative claire et simple [DF 87a], ce langage s'est révélé un outil puissant et aisé à manipuler tout au long de la construction de la spécification de PROLOG standard [DR 87].
Depuis quelques années, l'idée de coupler la programmation logique à la programmation concurrente a généré de nouveaux langages de programmation : Concurrent Prolog [Sha 86], Guarded Horn Clauses [Ued 85b], PARLOG [CG 86].
Dérivés de la programmation en clauses de Horn, ces langages cependant diffèrent de PROLOG par deux caractéristiques essentielles :
- ils sont non déterministes
- leur interprétation utilise les ressources du parallélisme **et** et **ou** et la gestion des communications entre processus s'effectue par le biais de nouveaux prédicats prédéfinis.

Actuellement, leur sémantique opérationnelle reste peu explorée ([SAR 86], [Beck 86], [Ued 85a]). Le développement au sein de notre équipe d'un compilateur PARLOG (dans le cadre du projet ESPRIT) nous a incité à écrire une spécification formelle de la sémantique opérationnelle de ce langage, en utilisant les méthodes mises en œuvre pour PROLOG standard.
La réalisation de cette spécification nous a permis de vérifier qu'il était tout à fait possible de décrire le non déterminisme inhérent à ce type de langages, et d'élargir le champ d'application de cette méthode de spécification.

Mots clés :

spécification formelle, sémantique opérationnelle, non déterminisme, parallélisme, programmation logique concurrente, sémantique de PARLOG

Note : cette recherche a été partiellement supportée par le Ministère de la Recherche et de l'Enseignement Supérieur, le GRECO-METHEOL et le projet COCOS#956 du programme ESPRIT.

Semantics of the Concurrent Logic Programming Language PARLOG

G. RICHARD - A. RIZK
INRIA
Domaine de Voluceau
B.P.105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36
uucp: mcvax!inria!ariane!richard
uucp: mcvax!inria!ariane!rizk

Abstract :

This paper presents a formal semantics specification for the concurrent logic programming language PARLOG. The semantics of this language which has so far remained unstudied is specified using a novel method based on Horn clauses with negation. The method employed could also be applied to other concurrent logic languages without any major difficulties.

Introduction

It is generally well known that the writing of programs destined for parallel execution is not a trivial task. The choice of the appropriate language and formalism is therefore a crucial problem : the semantics must be clear and the parallelism inherent to their structure. Logic programming languages and in particular Horn Clauses have these qualities. In fact, given a set of Horn Clauses, there exist a large number of resolution strategies, notably parallel strategies, different from that exploited in Standard Prolog. These strategies have given birth to a series of languages which do not in general preserve the purely declarative nature of Horn Clauses but rather add a dimension to their power of expression. PARLOG is one such language. Its operational semantics, however, has only so far been described in natural language by its designers [CG86].

We propose here therefore, a formal specification of the semantics of a PARLOG interpreter by using methods already experimented with a sequential deterministic language (Standard PROLOG [DF 87b], [DR 87]) but so far never applied to a concurrent nondeterministic one.

The underlying idea follows closely that described by Saraswat in the context of CP[SAR 86] and GHC[SAR 87] in that the semantical object considered is the well known AND/OR tree. The methodology for constructing the semantics however is aimed at those unfamiliar with the underlying theory of Plotkin's (state transition) method and, since it is described in first order predicate logic, is readily understandable by all those acquainted with the logic programming paradigm. The main contributions of this paper are therefore : firstly, it provides a formal semantics definition which takes into consideration the semantics community point of view **as well as** that of implementors and users; secondly, it provides the first formal definition of the PARLOG language.

In order to realize the first of these objectives, we make use of two novel concepts :

- a clear declarative semantics of Horn clauses **with negation**
- the notion of **relative denotation**

1) The PARLOG Language

Intrinsic to the resolution that governs the execution of logic programs [ROB 65], there are two principal sources of nondeterminism giving the potential for concurrent interpretation. Firstly, in the selection of an atom in a goal and secondly, in the selection of a clause. The two resulting forms of parallelism are termed **and** and **or** parallelism respectively.

PARLOG is the last of a family of similar languages that derives from the Relational Language[CG 81] and includes Concurrent Prolog [SHA 83] and Guarded Horn Clauses [Ued 85].

PARLOG [CG 84] is a concurrent programming language that exploits **and** parallelism with **and-stream communication** and **or** parallelism through the use of guarded Horn clauses and **don't care non-determinism.**

a) syntax of a PARLOG program

We give here a simplified syntax of PARLOG which is sufficient at present for the work described in this paper.

A PARLOG program consists of :

- a finite set of packets : a packet defining a predicate is a finite set of guarded Horn clauses. Each clause is separated from the following clause by either the **parallel search operator** denoted by '.' or the **sequential search operator** denoted by ';'.
- a set of associated mode declarations.

A guarded Horn clause has the form:

H <- G_1 op...op G_n : B_1 op...op B_m.

where H is an atom (clause head), G_1,..., G_n a set of atoms forming the guard and B_1,...,B_m a set of atoms forming the body of the clause.

The ':' symbol that separates the guard from the body is known as the **commit** operator.

The 'op' symbol stands for one of two operators : The **parallel conjunction** ',' or the **sequential conjunction** '&'.

The guard always exists but it may be reduced to **true** represented by an empty list.

A mode declaration associated with a predicate definition P of arity n takes the form :

$$\text{mode } P(m_1,..., m_n) .$$

where m_i is one of two symbols input or output denoted by ? and ^ in concrete syntax.

Note : the declaration of modes in PARLOG is associated with the predicate itself and not with the call or the clause as in Concurrent Prolog.

b) declarative semantics

The logic semantics of a guarded clause is the usual one of an unguarded Horn clause :

H <- $G_1,...,G_n, B_1,...,B_m$. where the guard forms an integral part of the clause body.

If G_1 and.... andG_n and B_1 and... and B_m then H .

We notice therefore, that within a clause, the declarative reading of the commit operator is that of conjunction. However, where there exist a number of clauses defining the same predicate a simple declarative reading is not sufficient to give an indication of the results given by an interpreter : an operational reading is therefore indispensable.

c) operational semantics

The operational semantics of PARLOG is completely different from that well known of a PROLOG interpreter. It is only at this stage that nondeterminism which characterises PARLOG intervenes.

We will give first some definitions :

Let t be a term : we denote by V(t) , the set of variables appearing in t.

Let s be a substitution and V a set of variables : We denote by s/V the restriction of s to V, i.e. the substitution defined by :

$$(s/V)(x) = s(x) \text{ if } x \leq V \text{ , else } (s/V)(x) = x.$$

Def 1 : input constraint

Given an atom $A = p(a_1, \ldots,a_n)$, and a clause head $H = p(t_1, \ldots,t_n)$ renamed with variables not appearing in A. We also denote the mode declaration associated with p by $p(m_1,\ldots,m_n)$.

Assuming that m_i is an input mode and s_i a unifier of a_i and t_i : we say that si satisfies the **input constraint** if and only if :

$s_i/V(a_i)$ is an empty substitution.

In other words, s_i does not instantiate the variables appearing in a_i. If by convention we denote dom(s), the domain of a substitution s, the preceding condition is equivalent to:

$$dom(s_i) \cap V(a_i) = \emptyset$$

Def 2 : suspension variables

If s_i doesn't satisfy the input constraint, this would mean : $dom(s_i) \cap V(a_i) \neq \emptyset$.

Any variable belonging to the set $dom(s_i) \cap V(a_i)$ is called **suspension variable.**

The set $\cup (dom(s_i) \cap V(a_i))$ for a_i describing the set of input arguments, is the **set of suspension variables.**

Def 3 : candidate clause

Given an atom $A = p(t_1,..., t_n)$: a clause defining the predicate p is a **candidate clause** at the resolution of A if and only if :

- The list of input terms in the clause head of this clause after renaming unifies with the list of input terms in A.
- The set of suspension variables is empty.
- The clause guard instantiated by the unifier allows an answer substitution.

Note that there are no constraints on the arguments specified as output in the mode declaration.

Examples :

mode p (?, ?,^)

A = p(f(X),X,Y)
H =p(f(1),Z,Z)
The unifier of the input term list is X<-1, Z<-1.
The input constraint is not satisfied by s1 : X<- 1. X is a suspension variable : such a unification will be suspended by the PARLOG interpreter on X.

A = p(f(X),X,1)
H = p(f(1), Z, 2): suspension on X for the same preceding reasons.
A = p(f(X),X,1)
H = p(f(1),Z,Z)
The unifier of the list of input terms is : Y<- X, Z <- X.
s1 is : Y<-X and s2 : Z<-X. There are therefore no suspension variables and unification is not suspended.

A = p(f(X), 1, Y)
H = p(f(Z), Z, T)
The unifier of the list of input terms is : X<-1, Z<-1
s1 is : Z<-X and s2 : Z<-1. There are no suspension variables and the unification is not suspended.

The functioning of a PARLOG interpreter could then be generally described as follows :

- Given a goal B, the resolution of atoms in B that are separated by the parallel conjunction is carried out in parallel and that of those separated by the sequential conjunction is carried out sequentially.
- For a given atom A, the search for a candidate clause at resolution of A is carried out in parallel for those clauses that are separated by the parallel search operator and serially for the clauses separated by the sequential search operator.
- Amongst all candidate clauses at the resolution of A, one and only one is selected for carrying on the resolution. The selection is performed **at random** if there are no sequential search operators, otherwise the choice is performed by respecting the rules of sequentiality specified by the occurences of the sequential operators : here the **don't care** nondeterminism is contrasted to the **don't know** nondeterminism of Prolog. The commit operator could therefore be considered as the concurrent counterpart of the PROLOG cut in the sense that it suppresses potential choices, whence the incompleteness of PARLOG interpreters.
- if there are no candidate clauses at the resolution of A, but there are clauses potentially useful not satisfying the input constraint (suspended clauses) then the resolution of A is suspended (not failed) until A is sufficiently instantiated so that one of the suspended clauses becomes candidate or else there are no suspended clauses left.

Mode constraints do not create supplementary failures. They only suspend the resolution of certain goals, thereby providing a means for synchronising PARLOG processes.

The sequential conjunction and sequential search operators constitute primitives for controlling the strategy of the interpreter.

d) properties of the PARLOG interpreter

correctness : it is clear that a PARLOG interpreter is correct in the sense that any solution it produces is a logical answer. This directly derives from the fact that a PARLOG interpreter is based on the SLD resolution. (This correctness is only theoretical in the sense that we ignore the problems of occur-check and the existence of impure primitives such as negation).

completeness : it is also obvious that a PARLOG interpreter is incomplete because:

- The choice of a candidate clause may generate an infinite loop that does not result in a solution whilst a logical solution might exist elsewhere.
- The choice of a candidate clause might carry on to a failure whilst another clause could have produced solutions.

We note 5 fundamental points distinguishing PARLOG from a standard PROLOG interpreter :

- PARLOG does not employ backtracking.
- PARLOG programs are not reversible due to the use of modes.
- PARLOG gives at most one answer amongst all possible answers.
- PARLOG does not necessarily give the same answer to the same program-goal couple.
- The operational semantics of PARLOG without sequential conjunctions and sequential search operators is independent of the order of clauses and the order of atoms in the body of the clause.

e) The notion of guard safety

Given a clause H <- G : B , its guard is termed **safe** if and only if, for any call A such that σ is the MGU of H and A with renaming, then any answer substitution s obtained from σ(G) is such that s/H is empty.

A clause is safe if its guard is safe and a program is safe if all its clauses are safe.

PARLOG specifies that all programs are safe and that this fundamental property is detectable at compile time.

The reason for this restriction is that it avoids the manipulation of supplementary environments which are necessary for storing multiple bindings before the execution of the commit operator.

2) The and/or tree, semantic object

a) the and/or tree

The **and/or** tree associated with a couple program-goal is a means of representing the resolution of problems [Kow 84]. This tree is well suited for the formal representation of the execution of Horn clauses for the following reasons:

- the **and/or** tree allows to show explicitly the different components of a goal which could effectively be treated in parallel by the interpreter (**and** parallelism).
- the various **or** branches corresponding to the potentially useful clauses for a goal resolution depict the potential for **or** parallelism.

Given a program P and a goal G, we can associate to them a unique **and/or** tree defined in the following manner:

- every node carries an atom
- if the node is an **or** node carrying the atom A, it has as many offsprings as there are clauses in P for which the head unifies with the atom A (eventually empty if such clauses do not exist). Each offspring is an **and** node.
- if the node is an **or** node carrying the atom A, then for each clause A0 <- A1,...,An of P whose head unifies with A with an MGU s, then there is an offspring **and** node that carries the atom s(A0) and possesses n **or** offsprings carrying respectively the atoms s(A1),..., s(An). if n = 0, (the case of an assertion), the **and** node is a leaf.
- finally, the root node is an **or** node.

algorithm for the construction of an and/or tree for a given couple program -goal

b : goal

P : program

T : and/or tree

P : = P ∪ { query <-b.}

T : = (query, or)

For each leaf (A,or) for which there exists in P a renamed clause head that unifies with A, do:

for each renamed clause A0 <- A1,..., An whose head unifies do:

s : = MGU(A, A0) ;

T : = T{(A, or) <- (A, or)

(s(A), and)

(s(A0), or) ... (s(An), or) }

The construction stops either when there are no more or leaves, or when the remaining or leaves do not have corresponding clauses.

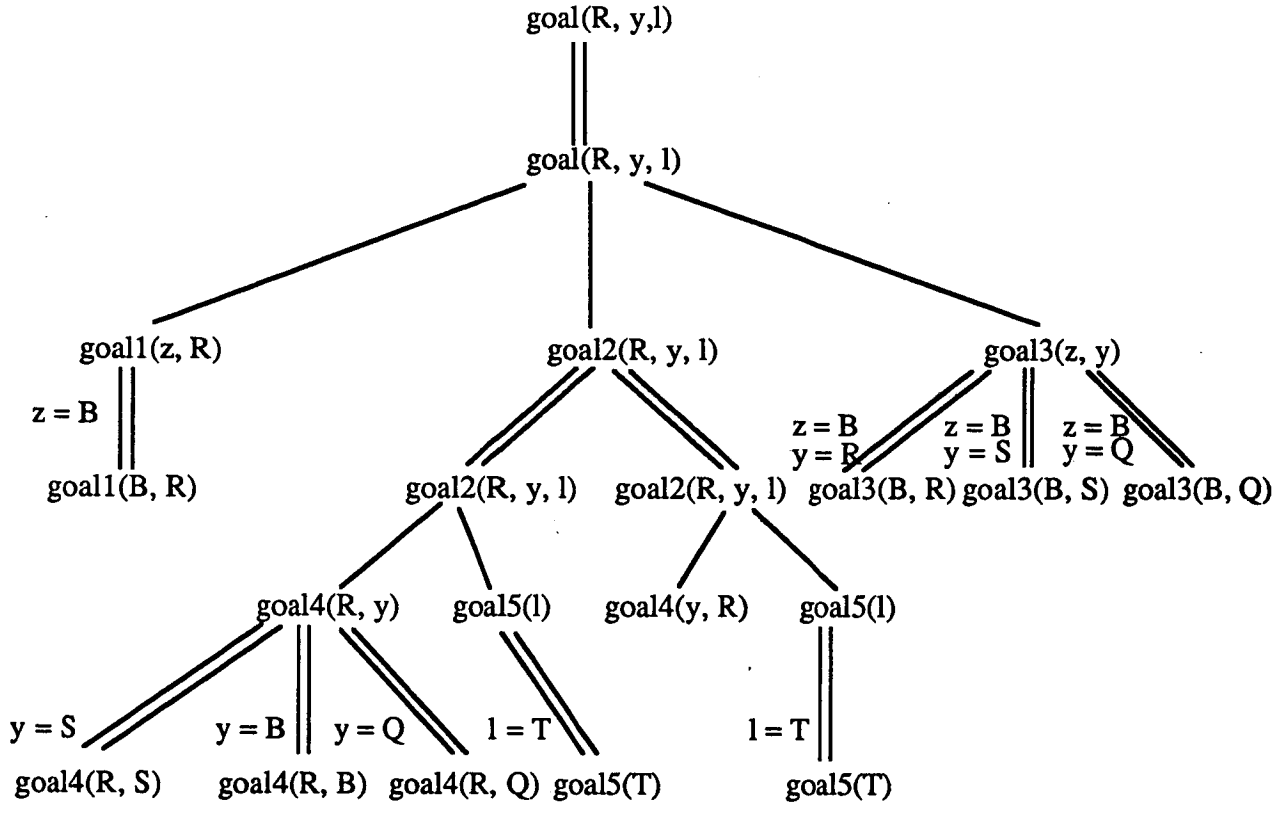
b) example

Consider an example : (variables in small letters, constants in capital)

goal(x, y, 1) <- goal1(z, x), goal2(x, y, 1), goal3(z, y).
goal1(A, Q).
goal1(B, R).
goal2(x, y, l) <- goal4(x, y), goal5(1).
goal2(x, y, 1) <- goal4(y, x), goal5(1).
goal3(B, R).
goal3(B, S).
goal3(B, Q).
goal4(R, S).
goal4(R, B).
goal4(R, Q).
goal5(T).

By convention, the branches deriving from an or node will be noted // and those deriving from an and node /.

The **and/or** tree corresponding to the query goal(R,y,l) is the following:



The substitutions are attached to the corresponding branches. The solutions are obtained by combining compatible substitutions. We see here two answer substitutions: y = S, l = T and y = Q, l = T.

c) semantics of PARLOG in terms of and/or tree

We can model the semantics of a Parlog interpreter to which we submit a couple program-goal as a construction of a subtree of the preceding **and/or** tree. The selection of a single clause amongst the candidate clauses can be modelled by the fact that each **or** node possesses at most one offspring node whilst the remaining nodes are left unchanged. The effect of the commit operator can be interpreted as the suppression of all **or** branches, except one, issuing from the same atom.

Moreover, because of the commit operator, the only **or** parallelism in PARLOG is situated at the level of guards evaluation. In other words, if two clauses H1 <- G1 : B1 and H2 <- G2 : B2 define the same predicate, there cannot exist two distinct processes working simultaneously on the evaluation of B1 and B2.

It should be noticed that this subtree is not unique : the selection of another candidate clause at a given time during execution results in the construction of a different subtree. This corresponds to the fact that a PARLOG interpreter does not give the same solution at different executions of a given couple program-goal. However, if a subtree is potentially constructible by an interpreter from a given couple program-goal and if the interpreter is **fair** then this subtree would eventually be constructed if we repeatedly submit to the interpreter the same couple program-goal a number of times. Therefore, **the set of potentially constructible subtrees** constitutes in fact the semantics of a couple program-goal with respect to the strategy of the PARLOG interpreter. Note that this implies that two failure executions are distinguished.

Consider the previous example in PARLOG :

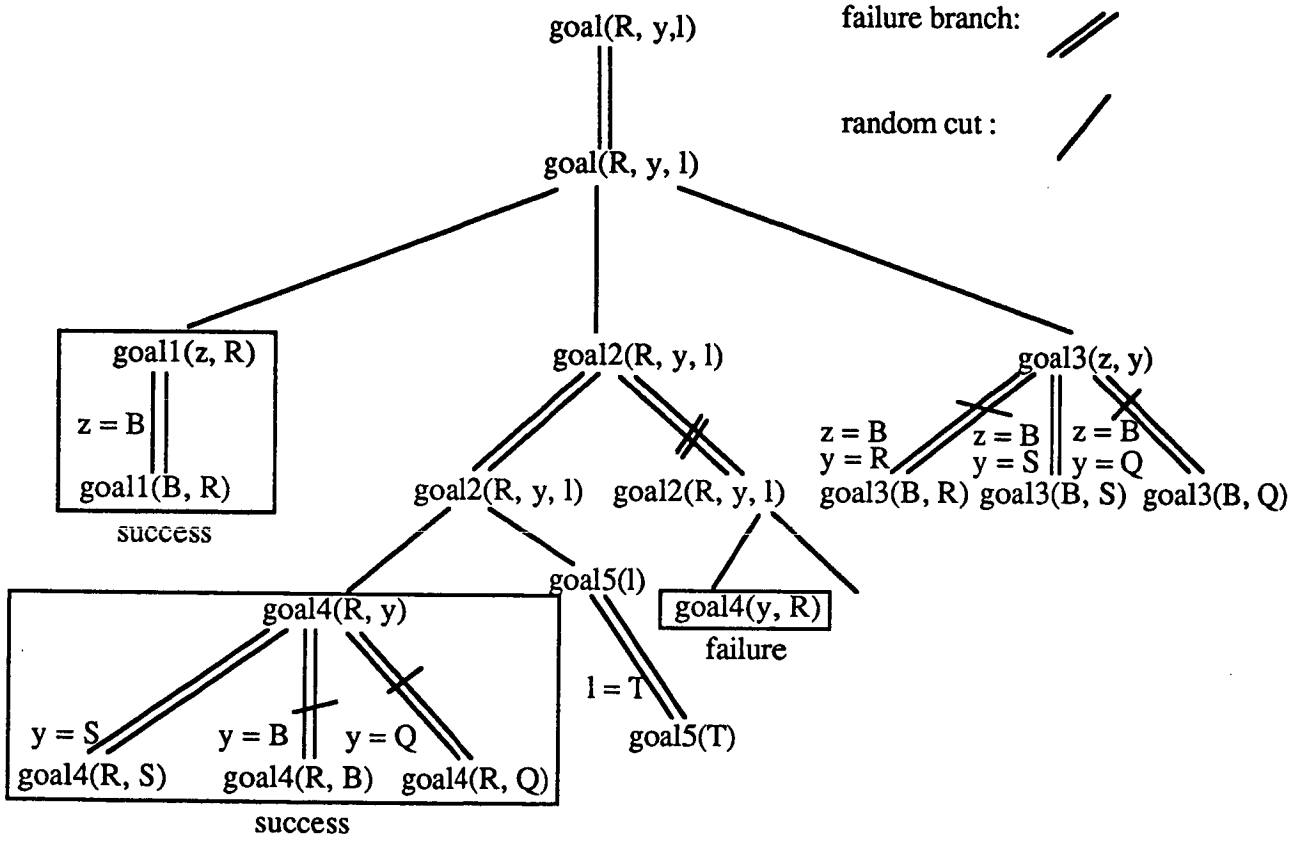
mode goal(?, ^, ^), mode goal1(^, ?), mode goal2(?, ?, ^), mode goal3(?, ^),

mode goal4(?, ^), mode goal5(^).

goal(x, y, l) <- goal1(z, x), goal2(x, y, l), goal3(z, y).

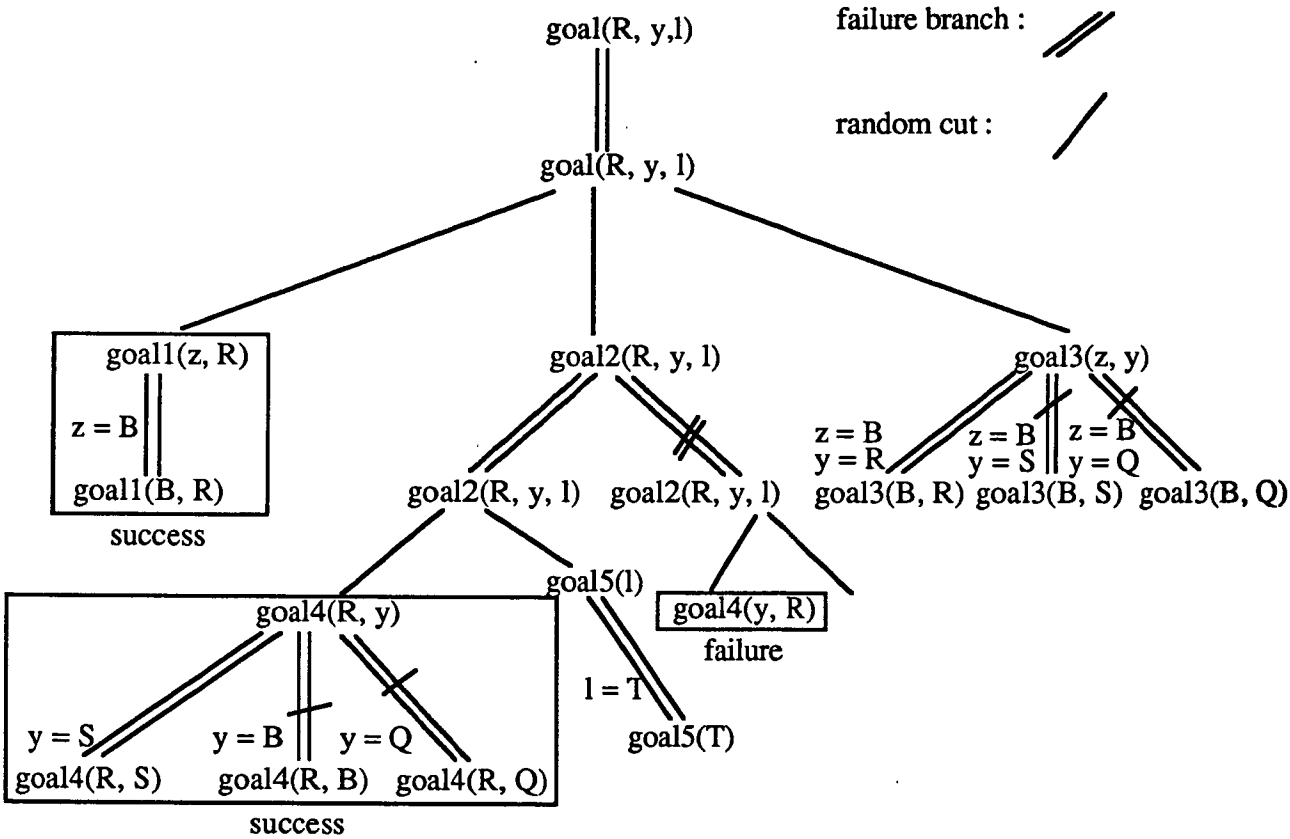
goal1(A, Q).
goal1(B, R).
goal2(x, y, l) <- goal4(x, y) : goal5(l).
goal2(x, y, l) <- goal4(y, x) : goal5(l).
goal3(B, R).
goal3(B, R).
goal3(B, Q).
goal4(R, S).
goal4(R, B).
goal4(R, Q).
goal5(T).

The framed subtrees denote the representation of guards. One tree might be :



The substitution provided by the PARLOG interpreter in this case is: y = S, l = T.

Another execution could produce the following tree where the choice of candidate clause for goal3 is different:



We see in this case that the PARLOG interpreter does not provide any solution (no compatible substitution in the tree).

Therefore, we describe the semantics of a PARLOG couple program-goal(P,G) submitted to an interpreter PARLOG as a relation (strictly a relation and not a function) linking the program and the goal to an associated **and/or** subtree T : **parlog-semantics**(P,G,T).

4) the formal specification

Two essential principles are kept in mind :

- **readability** of the specification: this can be realised by the choice of a specification language with clear semantics and with which it is easy to realise informal proofs.
- **minimality** of the specification : the specification must describe all the interesting properties of the concept and nothing more. This is achieved by the use of the notion of relative denotation.

a) the specification language and its semantics

As we mentioned earlier, we would need to use a formal language, but one that is readable and, as much as possible, modular. Therefore we choose here as a formalism, Horn clauses with negation in clause bodies. This formalism meets these requirements and has already been used for the formal specification of standard PROLOG [DR 87]. In addition, it possesses a purely declarative semantics which has been fully formalised in the work of [DF 87a] to which the reader is referred for a complete description. Accordingly, the specification language adopted has the following characteristics :

- it is a 1st order logic language with which we can write Horn clauses with negation.
- its semantics is purely declarative and based on the notion of proof trees with negation. The order of clauses and the order of atoms in the body are, therefore, completely irrelevant.
- its semantics is only defined for programs that are finitely founded. Our **parlog-semantics** program meets this requirement . More precisely, it is stratified in the sense of [ABW 86] and counts 4 levels of stratification.

b) proof methods

The specification will be even more comprehensible if it is possible, in an informal manner at least, to be convinced of two basic properties:

1) that the tree transformations that are described are the desired ones : this is a problem of correctness.
2) that all possible cases have been considered: this is a problem of completeness.

Proof methods of correctness and completeness have been developed in [Der87] and [DF87a]. Application of these methods necessitates :

- associating to each predicate in the specification an **assertion of correctness** and an **assertion of completeness** . In all cases, if P is the predicate under consideration and X, Y a partition of the set of its arguments, then the correctness assertion is of the type : **if** A(X) **then** B(X, Y) and the assertion of completeness of the type : A(X) **and** B(X, Y) .
- In the case of negative use of P, the correctness assertion is of the type : **if** A(X) **then** not B(X,Y).

However, since we do not dispose of a "specification" of our specification, these assertions are written in natural language and should be read as comments. These assertions allow to understand how the clauses are constructed: in fact, by assuming the validity of the correctness assertions for the atoms in the clause body, we demonstrate that the assertion relative to the head atom is valid.

Note finally that the writing of the specification is more guided by simplicity of proofs than by elegance. For this reason, we might notice some introduced redundancy which in our point of view improves readability.

c) relative denotation

Given a (possibly infinite) set of atoms, E : we can associate to it the following program E'={A<-. / A in E }. Hence we define the denotation of a program P relative to E to be the denotation of the program P $\cup$ E'. The program E' is the simplest program whose denotation is E, in the sense that it contains only assertions.

When we want to have a well defined set of atoms in the denotation of our program, then the use of the denotation of P relative to this set is a means of avoiding many problems :

- on one hand, the choice of a semantic representation for the objects we want to specify: this is the case of variables in PARLOG, of substitutions, of atoms etc...
- on the other hand, the writing of a complete description of this set in Horn clauses: the case of unification, renaming of clauses, directional unification.etc...

In particular, the appropriate use of relative denotation avoids having a Horn clause specification of those concepts that are obvious and implementation independent. In fact, an overspecification of such concepts runs the danger of being too restraining on implementers. This case is obvious for concepts like integers and arithmetic operators.

5) introduction to the reading of the specification

a) data structures

- in order to represent the **and/or** tree we use a forest structure : a forest might be viewed as a list of trees (a tree being a special case of forest). This structure simplifies the writing of recursive clauses where it takes part. It is represented with the aid of a function symbol **for**, with three arguments : the first one represents a node, the other two are forests.
- to represent the nodes of the **and/or** tree, we use a function symbol, **nd**, with four arguments: the first argument is the unique number of the node (numbering according to Dewey) the second argument represents the atom carried by the node and the third, a list of variables on which the node is suspended: an empty list, therefore, indicates that the node is not suspended. The fourth argument is a flag that indicates whether the node is in wait state (due to a sequential conjunction) or not.
- to represent a clause, we use a function symbol, **cl**, with four arguments : the first argument represents the clause search operator separating the clause from the preceding one in the program (the first clause of each predicate is always associated to the parallel search operator); the second argument represents the clause head; the third represents the guard, and the fourth argument represents the body.
- a program is simply a list of predicates.
- a predicate is a couple of packet and associated modes declaration.
- a packet is a list of clauses that takes into account the parallel and sequential search operators.

b) syntax of the specification language

The specification language is a 1st order logic language that consists of :

- ♦ an enumerated set of variables denoted with the Edinburgh syntax.
- ♦ an enumerated set of function symbols including:
  - arity 0 : true, fail, nil, vid, root, 0, wait, no-wait
  - arity 1 : s (for natural numbers)
  - arity 2 : . (for lists), seq(sequential search), par(parallel search), pred
  - arity 3 : for (for forest), cl (for clauses)
  - arity 4 : nd (for nodes)
- ♦ a set of around sixty symbols for predicates defined by clauses : eg **suspended#2**, **addchild#4**, etc...
- ♦ a set of 14 symbols of predicates not defined with clauses and prefixed with an **L** eg. **L-unify#5**, etc...
- ♦ three logic operators : **not**, $\Leftarrow$, ,.

With this language, we write Horn clauses with negation : our whole program is stratified (4 levels of stratification).

c) the relation parlog-semantics

The essential part of the specification consists of defining the relation **parlog-semantics** which shows that an **and/or** tree belongs to the semantics of a couple program-goal(P, G), i.e. it is potentially constructible by the PARLOG interpreter from (P, G).

This relation is defined mostly by the relation **build-and/or-tree** which specifies the conditions for the construction of an **and/or** tree given the tree of the preceding step. There are several cases here : either the tree of the preceding step is a success tree, a suspended tree, or it is a failure tree in which case the construction terminates, otherwise construction continues.

The relation **treatment** therefore specifies the evolution of the **and/or** tree step by step, depending on whether or not there exist candidate clauses for the atom under consideration.

6) discussion

a) pros and cons of the methodology

A formal specification is written to be used by the language implementers, users and all those interested in analysing the language concepts (program equivalence, program properties, etc...).

Our specification naturally possesses the properties of the language it is written in : readability, modularity. We therefore consider that the main objective of the specification has been achieved: any person familiar with PARLOG should be able to understand the specification without any *extra* knowledge.

Moreover, we can easily derive from it an executable specification with control over the adequateness between the formal version and the runnable one. That is to say, by the way of proof methods about operational properties of logic programs, we can test that the denotation of our formal specification is the same as the runnable one. These methods are explained in detail in [DM88].

Finally, this specification method allows, without any major difficulties, the treatment of predicates and operators whose semantics is purely operational. This has been proven by the treatment of sequential conjunction and sequential search operators of PARLOG. For all these reasons, our specification seems well suited to the needs of implementers and users.

Certain program properties could be expressed simply in terms of **and/or** subtrees. For example, a potential deadlock is expressed by the fact that there exists a completely suspended tree in the semantics :

deadlock (P, G) <- **parlog-semantics** (P, G, F),
is-a-suspended-tree (F).

Meanwhile, the problem of program comparisons from **and/or** trees remains open : however, the nature of our specification seems to lend itself readily to the development of automatic tools for the derivation of proofs.

b) related work

Our specification, though using the same language as Pereira and Monteiro [PM 78] does not describe the functioning of the interpreter, and in this sense seems of higher level.

Ueda [Ued 85] gives, in natural language, a resolution algorithm for GHC. Although this method is rigorous and easily comprehensible, the problems inherent to this kind of method remain (most importantly the risk of ambiguity).

Beckmann [Beck 86] uses a mapping towards CCS (Calculus for Communicating Systems [Mi 80]) and Saraswat [Sar 86] uses the Plotkin methodology [Plot 81]. They construct a formal semantics for concurrent logical languages which seem quite adapted to problems of proofs of program properties and equivalence.

In particular, Saraswat[SAR 87] uses the structural operational semantics of Plotkin to define the operational semantics of GHC. His semantics is described in terms of transitions over configurations which define the system evolution. Although it is not directly obvious from the transistion rules, the underlying described notions are the proof tree and parallel input resolution, which is rather close to our notion of partial AND/OR tree.

Saraswat gives a complete description of the unification in GHC. In our case, we consider it as an atomic action, although we believe it could be easily added to our specification. The main advantage of Saraswat's method is that his formalisation makes possible, in a very elegant way, to prove properties about programs.

However, an executable specification can be more easily derived from our specification than from Plotkins structural semantics, and also more easily proven to possess the same denotation.[DM 88].

c) future work

As to PARLOG, we have not evoked the problem of looping guards which seems to be closely related to the fairness property of the interpreter. The specification of similar languages (GHC, Concurrent Prolog) and eventually other kind of concurrent logical languages seems easily treatable using the same method and should not prove too different from the current one for PARLOG.

Conclusions

This paper described a formal specification of the complete semantics of PARLOG - the only semantics existing until now is the informal one of its designers [CG 86]. Except for a few additional predefined predicates of the language that would necessitate slight extensions, we consider having treated the essential bulk of PARLOG and the important features of the other languages of the same family.

Logic programming with Horn clauses plus negation, having a purely declarative semantics [Fer 87] and coupled with correctness and completeness proof methods, seems to be a powerful tool for the specification of logic languages, both sequential [DR 87] and parallel.

Acknowledgements

We are indebted to C. and P. Codognet, P. Deransart, G. Ferrand and M. Jourdan for the discussions we had together. This work was partially undertaken within the framework of the European ESPRIT project 951 "COCOS" and the METHEOL group of "GRECO de programmation du CNRS".

References

[ABW 86] APT K.R., BLAIR H. and WALKER A. : Towards a Theory of Declarative Knowledge, Res. Report 86-10 LITP Paris 1986.

[Beck 86] BECKMANN L. : Towards a formal semantics for concurrent logic programming languages, Third Int. Conf. on Logic Programming, London 1986.

[CG 86] CLARK K., GREGORY S. : PARLOG : Parallel Programming on Logic, ACM transactions on Programming Languages and Systems, Vol 8 n° 1, January 86, pp 1-49.

[Der 87] DERANSART P. : Partial Correctness of Logic programs (Preuves de Correction Partielle des Programmes Logiques).

[DF 87a] DERANSART P., FERRAND G. : Programmation en Logique avec Négation, Rapport de recherche n° 87-3, Université d'Orléans (LIFO) 1987.

[DF 87b] DERANSART P., FERRAND G. : An Operational Formal Definition of PROLOG, Fourth IEEE symposium on Logic Programming, August 24 - September 4, 1987, San Francisco.

[DM 88] DERANSART P., MALUSJINSKI j. : A Grammatical view of Logic Programming, PLILP 88, Orléans, May 1988.

[DR 87] DERANSART P. RICHARD G. : Formal specification of PROLOG, third draft - December 1987, PS 210 BSI/AFNOR publication.

[FT 86] FURUKAWA K., TAKEUCHI A. : Parallel Logic Programming Languages, Lecture Notes in Computer Science 225, LPC London 14-18 July 1986.

[Gre 87] GREGORY S. : Parallel Logic Programming in PARLOG, Addison Wesley.

[Kow 84] KOWALSKI R. : Logic for Problem Solving (5ième édition), North Holland.

[Plot 81] PLOTKIN G.O. : A structural approach to operationnal semantics, DAIMI FN-19, CS Department, University of Aarhus, September 1981.

[PM 78] PEREIRA L.M., MONTEIRO L.F. : The Semantics of parallelism and co-routining in Logic Programming, in Proc. of Colloquia Mathematica Societatis Janos Bolyai pp 611-657, 1978.

[Rob 65] ROBINSON J. A. : A Machine Oriented Logic based on the Resolution Principle, J. ACM Vol 12 n° 1, 1965.

[Sar 86] SARASWAT A. : The Concurrent Logic Programming Language CP : definition and operational semantics, September 86.

[Sar 87] SARASWAT A. : GHC : Operational Semantics, Problems and Relationship with CP(↓, |). Symposium of Logic Programming, San Francisco, 1987.

[Sha 86] SHAPIRO E.Y. : Concurrent Prolog : a progress report, IEEE Computer, August 1986.

[Ued 85a] UEDA K. : On the Operational Semantics of G.H.C., ICOT Technical Memorandum : TM-0136.

[Ued 85b] UEDA K. : Guarded Horn Clauses - ICOT technical report TR-103 June 1985.

PARLOG SEMANTICS

DEFINITION OF THE DATA STRUCTURES

The and/or tree is represented by a forest

forest definition

```
is-a-forest( vid ).
is-a-forest( for( N, F1, F2 ))
      ⇐  is-a-node( N ),
         is-a-forest( F1 ),
         is-a-forest( F2 ).
```

node definition

```
is-a-node( nd( I, A, L, W ))
      ⇐  is-a-dewey-number( I ),
         L-literal( A ),
         is-a-list-of-variables( L ),
         is-a-waiting-marker( W ).
```

dewey number definition

```
is-a-dewey-number( nil ).
is-a-dewey-number( X.L )
      ⇐  is-a-natural( X ),
         is-a-dewey-number( L ).
```

natural definition

```
is-a-natural( 0 ).
is-a-natural( s(X))
      ⇐  is-a-natural( X ).
```

list-of-variables definition

```
is-a-list-of-variables( nil ).
is-a-list-of-variables( X.L )
      ⇐  L-variable( X ),
         is-a-list-of-variables( L ).
```

waiting-marker definition

```
is-a-waiting-marker( wait ).
is-a-waiting-marker( no-wait ).
```

ABSTRACT SYNTAX

program definition

```
is-a-program( P )
      ⇐  is-a-list-of-predicate( P ).
```

list-of-predicate definition

```
is-a-list-of-predicate( nil ).
is-a-list-of-predicate( P.Q )
      ⇐  is-a-predicate( P ),
         is-a-list-of-predicate( Q ).
```

predicate definition

```
is-a-predicate( pred( Q, M) )
      ⇐  is-a-packet( Q ),
         L-mode( Q,M ).
```

packet definition

```
is-a-packet( C )
      ⇐  is-a-clause( C ).
is-a-packet( seq( Q, P ) )
      ⇐  is-a-packet( Q ),
         is-a-packet( P ),
         L-same-name( Q, P ).

is-a-packet( par( Q, P ) )
      ⇐  is-a-packet( Q ),
         is-a-packet( P ),
         L-same-name( Q, P ).
```

clause definition

```
is-a-clause( cl( H, G, B ))
      ⇐  L-literal ( H ),
         is-a-guard ( G ),
         is-a-body ( B ).
```

guard definition

```
is-a-guard ( G )
      ⇐  is-a-body ( G ).
```

body definition

```
is-a-body( A ).
      ⇐  L-literal( A ).
is-a-body ( fork( A, B ) )
      ⇐  is-a-body( A ),
         is-a-body ( B ).
is-a-body ( and( A, B ) )
      ⇐  is-a-body ( A ),
         is-a-body ( B ).
```

PRINCIPAL PART

parlog-semantics#3 :
parlog-semantics(P, G, F) then if P is a program and G a goal then F is a partial and/or tree obtained by a constructive tree-walk of the and/or tree reduced to root.

```
parlog-semantics( P, G, F )
      ⇐  node N is nd( nil, root, nil, no-wait ),
         build-and/or-tree(pred(cl(root,true,G),nil).P,
                                   for(N,vid,vid),F)
```

build-and/or-tree#3 :
build-and/or-tree(P, F, F1) then if P is a program, F a partial and/or tree then F1 is a partial and/or tree obtained by a constructive tree-walk starting with the tree F.

```
build-and/or-tree( _ , F, F )
      ⇐   is-a-success-tree ( F ).

build-and/or-tree( _ , F, F )
      ⇐   is-a-suspended-tree ( F ).

build-and/or-tree( _ , F, F )
      ⇐   is-a-failure-tree ( F ).

build-and/or-tree( P, F, F1)
      ⇐   not   is-a-failure-tree ( F ),
          active-leaves( F , L ),
          L-first-active-leaf ( L, N ),
          treatment( P, F, N, F1 ).

build-and/or-tree( P, F, F1)
      ⇐   not   is-a-failure-tree ( F ),
          active-leaves ( F , L ),
          L-first-active-leaf ( L, N ),
          treatment( P, F, N, F2 ),
          build-and/or-tree( P, F2, F1).
```

is-a-success-tree#1 :
is-a-success-tree(F) then if F is a and/or tree then this is a success tree.

```
is-a-success-tree ( F ) ⇐ success-leaves-only( F ).
```

is-a-suspended-tree#1 :
is-a-suspended-tree(F) then if F is a and/or tree then this is a suspended tree.

```
is-a-suspended-tree ( F ) ⇐ suspended( F, _ ).
```

is-a-failure-tree#1 :
is-a-failure-tree(F) then if F is a and/or tree then this is a failure tree.

```
is-a-failure-tree ( F ) ⇐ has-a-failure-node( F ).
```

success-leaves-only#1 :
success-leaves-only(F) then if F is a and/or tree then all its leaves are success leaves.

```
success-leaves-only ( F )
      ⇐   leaves( F, L ),
          all-success-nodes( L ).
```

all-success-nodes#1 :
all-success-nodes(L) then if L is a list of nodes then all the nodes are success nodes.

```
all-success-nodes ( nil ).

all-success-nodes ( N.L )
      ⇐   is-a-success-node( N ),
          all-success-nodes( L ).
```

is-a-success-node#1:
is-a-success-node(N) then if N is a node then this is a success node.

```
is-a-success-node( nd( _ , true, nil , no-wait ) ).
```

suspended#2 :
suspended(F, V) then if F is a partial and/or tree then all the non success leaves of F are suspended on the list of variables V and it exists at least a suspended leaf or are leaves in wait state.

```
suspended( F, V )
      ⇐   leaves( F, L ),
          list-of-susp/wait/success-nodes( L ),
          set-of-suspended-nodes( L, S, V ),
          not  equal ( S, nil ).
```

list-of-susp/wait/success-nodes#1 :
list-of-susp/wait/success-nodes(L) then if L is a list of nodes then it only contains nodes that are suspended, waiting or success.

```
list-of-susp/wait/success-nodes( nil ).

list-of-susp/wait/success-nodes( N.L )
      ⇐   is-a-suspended-node( N ),
          list-of-susp/wait/success-nodes(L).

list-of-susp/wait/success-nodes( N.L )
      ⇐   is-a-waiting-node( N ),
          list-of-susp/wait/success-nodes(L).

list-of-susp/wait/success-nodes( N.L )
      ⇐   is-a-success-node( N ),
          list-of-susp/wait/success-nodes(L).
```

set-of-suspended-nodes# 3 :
set-of-suspended-nodes(L, S, V) then if L is a list of nodes then S is a list of nodes of L which are suspended and these nodes are suspended on the list of variables V.

```
set-of-suspended-nodes( nil, nil, nil ).

set-of-suspended-nodes( N.L, S, V )
      ⇐   not  is-a-suspended-node( N ),
          set-of-suspended-nodes( L, S, V ).

set-of-suspended-nodes( N.L, N.S, V.V1 )
      ⇐   suspended-node-on-var( N, V ),
          set-of-suspended-nodes( L, S, V1 ).
```

is-a-suspended-node#1:
is-a-suspended-node(N) then if N is a node then it is suspended.

```
is-a-suspended-node( N )
      ⇐   suspended-node-on-var( N, _ ).
```

suspended-node-on-var#2:
suspended-node-on-var(N, V) then if N is a node then N is suspended on the list of variables V.

```
suspended-node-on-var( nd( _ , _ , V, _ ), V )
        ⇐   not  equal( V, nil ).
```

is-a-failure-node#1:
is-a-failure-node(N) then if N is a node then it is a failure node .

```
is-a-failure-node( nd( _ , fail, nil, no-wait ) ).
```

is-a-waiting-node#1:
is-a-waiting-node(N) then if N is a node then N is in wait state.

```
is-a-waiting-node( nd( _ , _ , _ , wait ) ).
```

has-a-failure-node#1:
has-a-failure-node(F) then if F is a tree then it possesses a failure node

```
has-a-failure-node( F )
        ⇐   leaves( F, L ),
            member( N, L ),
            is-a-failure-node( N ).
```

active-leaves#2:
active-leaves(F, L) then if F is a tree then L is the list of active nodes of F.

```
active-leaves( F, L )
        ⇐   leaves( F, L1),
            set-of-active-nodes( L1, L ).
```

set-of-active-nodes#2 :
set-of-active-nodes(L, L1) then if L is a list of nodes then L1 is the list of active nodes in L.

```
set-of-active-nodes( nil,  nil ).

set-of-active-nodes( N.L, N.L1 )
        ⇐   is-an-active-node( N ),
            set-of-active-nodes( L, L1 ).

set-of-active-nodes( N.L, L1 )
        ⇐   not   is-an-active-node( N ),
            set-of-active-nodes( L, L1 ).
```

is-an-active-node#1:
is-an-active-node(N) then if N is a node then it is an active node.

```
is-an-active-node( N )
        ⇐   not  is-a-success-node( N ),
            not  is-a-suspended-node( N ),
            not  is-a-failure-node( N ),
            not  is-a-waiting-node( N ).
```

treatment#4 :
treatment(P, F, N, F1) then if P is a program, F a partial and/or tree , N the active leaf of F then F1 is the and/or tree obtained after treatment of N.

```
treatment( P, F, N, F1 )
        ⇐   goal( N , A ),
            L-packet( P, A , M, Q ),
            set-of-useful-clauses( P, F, A , M, Q, L ),
            not  equal( L, nil ),
            L-random-choice  ( L, C ),
            buildtree( P, F, N, C, S, T ),
            instantiation( F, S, F2 ),
            addforest  ( F2, N, T, F3 ),
            last-leaf( T, N1 ),
            update-waiting-marker( F3, N1, F1 ).

treatment( P, F, N, F1 )
        ⇐   goal( N , A ),
            L-packet( P, A , M, Q ),
            set-of-useful-clauses( P, F, A , M, Q , L ),
            not  equal( L, nil ),
            L-random-choice  ( L, C ),
            not  possible-tree( P, F, N, C ),
            nodenumber( N, I ),
            node N1 is nd( 0.I, fail, nil, no-wait ),
            addchild  ( F, N, N1, F1 ).

treatment( P, F, N, F1 )
        ⇐   goal( N , A ),
            L-packet( P, A , M, Q ),
            set-of-useful-clauses( P, F, A, M, Q, nil ),
            set-of-suspension( P, F, A, M,Q,V ),
            not  equal( V, nil ),
            suspension( F, N, V, F1 ).

treatment( P, F, N, F1 )
        ⇐   goal( N , A ),
            L-packet( P, A , M, Q ),
            set-of-useful-clauses( P, F, A , M, Q , nil ),
            set-of-suspension( P, F, A, M, Q, nil ),
            nodenumber( N, I ),
            node N1 is nd( 0.I, fail, nil, no-wait ),
            addchild  ( F, N, N1, F1 ).
```

update-waiting-marker#3 :
update-waiting-marker(F, N, F1) then if F is a and/or tree and N a leaf then F1 is the and/or tree after updating the waiting node.

```
update-waiting-marker( F, N, F )
        ⇐   leaves( F , L ),
            waiting-leaves( L, N, nil ).

update-waiting-marker( F, N, F )
        ⇐   leaves( F , L ),
            first-waiting-node( L, N, N1 ),
            preceding-tree( F, N1, T ),
            not   is-a-success-tree( T ).

update-waiting-marker( F, N, F1 )
        ⇐   leaves( F , L ),
            first-waiting-node( L, N, N1 ),
            preceding-tree( F, N1, T ),
            is-a-success-tree( T ),
            ready( F, N1, F1 ).
```

waiting-leaves#3 :
waiting-leaves(L, N, L1) then if L is a list of nodes and N a node in L then L1 is the list of nodes of L situated after N that are waiting.

```
waiting-leaves( N.nil, N, nil ).

waiting-leaves( N.N1.L, N, N1.L1 )
      ⇐ is-a-waiting-node( N1 ),
        waiting-leaves( N1.L1, N1, L1 ).

waiting-leaves( N.N1.L, N, L1 )
      ⇐ not is-a-waiting-node( N1 ),
        waiting-leaves( N1.L1, N1, L1 ).

waiting-leaves( M.L, N, L1 )
      ⇐ not equal( N, M ),
        waiting-leaves( L, N, L1 ).
```

first-waiting-node#3 :
first-waiting-node(L, N, N1) then if L is a list of nodes and N a node of L then N1 is the first waiting node situated after N in L.

```
first-waiting-node( L, N, N1 )
      ⇐ waiting-leaves( L, N, N1._ ).
```

set-of-useful-clauses#6 :
set-of-useful-clauses(P, F, A, M, Q, L) then if P is a program, F is a partial and/or tree and A an atom, Q a packet of clauses defining the atom A and M the mode declaration associated then L is the list of clauses of Q which are candidates at the resolution of A in the context of F and which could be chosen for the resolution of A following the sequential search operator.

```
set-of-useful-clauses( P , F,  A, M, C, C.nil )
      ⇐ is-a-candidate-clause ( P, F, A, M, C ).

set-of-useful-clauses( P , F,  A,  M, C, nil )
      ⇐ is-a-clause ( C ),
        is-not-a-candidate-clause ( P, F, A, M, C ).

set-of-useful-clauses( P , F,  A, M, seq( C, Q ), C.nil )
      ⇐ is-a-candidate-clause ( P, F, A, M, C ).

set-of-useful-clauses( P , F,  A, M, seq( C, Q ), L )
      ⇐ is-a-clause ( C ),
        is-not-a-candidate-clause ( P, F, A, M, C ),
        set-of-useful-clauses  ( P, F, A, M, Q, L ).

set-of-useful-clauses( P , F,  A, M, seq( Q1, Q2 ), L )
      ⇐ not is-a-clause ( Q1 ),
        set-of-useful-clauses  ( P, F, A, M, Q1, L ),
        not equal( L, nil ).

set-of-useful-clauses( P , F,  A, M, seq( Q1, Q2 ), L )
      ⇐ not is-a-clause ( C ),
        set-of-useful-clauses  ( P, F, A, M, Q1, nil ),
        set-of-useful-clauses  ( P, F, A, M, Q2, L ).

set-of-useful-clauses( P , F,  A, M, par( C, Q ), C.L )
      ⇐ is-a-candidate-clause ( P, F, A, M, C ),
        set-of-useful-clauses  ( P, F, A, M, Q, L ).

set-of-useful-clauses( P , F,  A, M, par( C, Q ), L )
      ⇐ is-a-clause ( C ),
        is-not-a-candidate-clause ( P, F, A, M, C ),
        set-of-useful-clauses  ( P, F, A, M, Q, L ).

set-of-useful-clauses( P , F,  A, M, par( Q1,Q2 ), L )
      ⇐ not is-a-clause ( Q1 ),
        set-of-useful-clauses  ( P, F, A, M, Q1, L1 ),
        set-of-useful-clauses  ( P, F, A, M, Q2, L2 ),
        append( L1, L2, L ).
```

set-of-suspension#6:
set-of-suspension(P, F, A, M, Q, V) then if P is a program, F a partial and/or tree and A an atom, Q a packet of clauses defining the atom A and M the associated mode declaration then V is the list of suspended variables for the resolution of A.

```
set-of-suspension( P , F,  A, M, C, V.nil )
     ⇐ suspended-clause-on-var ( P, F, A, M, C, V ).

set-of-suspension( P , F,  A, M, C, nil )
     ⇐ is-a-clause ( C ),
       is-not-a-suspended-clause ( P, F, A, M, C ).

set-of-suspension( P , F,  A, M, seq( C, Q ), V.L )
     ⇐ suspended-clause-on-var( P, F, A, M, C, V ),
       set-of-suspension( P , F,  A, M,  Q, L ).

set-of-suspension( P , F,  A, M, seq( C, Q ), L )
     ⇐ is-a-clause ( C ),
       is-not-a-suspended-clause ( P, F, A, M, C ),
       set-of-suspension  ( P, F, A, M, Q, L ).

set-of-suspension( P , F,  A, M, seq( Q1, Q2 ), L )
     ⇐ not is-a-clause ( Q1 ),
       set-of-suspension  ( P, F, A, M, Q1, L1 ),
       set-of-suspension  ( P, F, A, M, Q2, L2 ),
       append( L1, L2, L ).

set-of-suspension(P , F,  A, M, par( C, Q ), V.L )
     ⇐ suspended-clause-on-var ( P, F, A, M, C, V ),
       set-of-suspension( P , F,  A, M,  Q, L ).

set-of-suspension( P , F,  A, M, par( C, Q ), L )
     ⇐ is-a-clause ( C ),
       is-not-a-suspended-clause ( P, F, A, M, C ),
       set-of-suspension  ( P, F, A, M, Q, L ).

set-of-suspension( P , F, A, M, par( Q1,Q2 ), L )
     ⇐ not is-a-clause ( Q1 ),
       set-of-suspension  ( P, F, A, M, Q1, L1 ),
       set-of-suspension  ( P, F, A, M, Q2, L2 ),
       append( L1, L2, L ).
```

is-a-candidate-clause#5 :
is-a-candidate-clause(P, F, A, M, C) then if P is a program, F a partial and/or tree , A an atom and M the associated mode declaration then C is a candidate clause for the resolution of A in the context of F.

```
is-a-candidate-clause( P, F, A, cl( H, G, _ ) )
    ⇐ L-rename ( F, cl( H, G, _ ), cl( H1, G1, _ )),
      L-input-unify ( A, H1, M, S, nil ),
      instantiate-body ( G1, S, G2 ),
      complete-parlog-semantics ( P, G2, F1 ),
      is-a-success-tree ( F1 ).
```

is-not-a-candidate-clause#5 :
is-not-a-candidate-clause(P, F, A, M, C) then if P is a program, F a partial and/or tree , A an atom and M the associated mode declaration then C is not a candidate clause for the resolution of A in the context of F.

```
is-not-a-candidate-clause( P, F, A, M, cl( H, _ , _ ) )
    ⇐ L-rename ( F, cl( H, _ , _ ), cl( H1, _ , _ ) ),
      not input-unifiable( A, H1, M ).

is-not-a-candidate-clause( P, F, A, M, cl( H, _ , _ ) )
    ⇐ L-rename ( F, cl( H, _ , _ ), cl( H1, _ , _ )),
      L-input-unify ( A, H1, M, S, V ),
      not equal ( V, nil ).

is-not-a-candidate-clause( P, F, A, M, cl( H, G, _ ) )
    ⇐ L-rename ( F, cl( H, G, _ ), cl( H1, G1, _ )),
      L-input-unify ( A, H1, M, S, nil ),
      instantiate-body ( G1, S, G2 ),
      complete-parlog-semantics ( P, G2, F1 ),
      not    is-a-success-tree ( F1 ).
```

instantiate-body#3 :
instantiate-body(B, S, B1) then if B is a goal and S a substitution then B1 is the goal B instanciated by S.

```
instantiate-body( A, S , A1 )
    ⇐ L-instance( A , S, A1 ).

instantiate-body(fork ( A, B ), S, fork( A1, B1))
    ⇐ instantiate-body( A , S, A1 ),
      instantiate-body( B, S, B1 ).

instantiate-body(and ( A, B ), S, and( A1, B1 ) )
    ⇐ instantiate-body( A , S, A1 ),
      instantiate-body( B, S, B1 ).
```

input-unifiable#3 :
input-unifiable(X, Y, M) then if X and Y are terms and M an associated mode declaration then their input arguments are unifiable.

```
input-unifiable( X, Y, M )
    ⇐ L-input-unify( X, Y, M, _ , _).
```

suspended-clause-on-var-#6 :
suspended-clause-on-var(P, F, A, M, C, V) then if P is a program, F a partial and/or tree , A an atom and M an associated mode declaration then C is a suspended clause on the set of variables V for the resolution of A in the context of F.

```
suspended-clause-on-var( P, F, A, M, cl( H, G, _ ), V )
    ⇐ L-rename ( F, cl( H, G, _ ), cl( H1, G1, _ ) ),
      L-input-unify ( A, H1, M, S, V ),
      not equal ( V, nil ),
      instantiate-body ( G1, S, G2 ),
      complete-parlog-semantics ( P, G2, F1 ),
      is-a-success-tree( F1 ).

suspended-clause-on-var(P, F, A, M, cl( H, G, _ ),V.V1)
    ⇐ L-rename ( F, cl( H, G, _ ), cl( H1, G1, _ ) ),
      L-input-unify ( A, H1, M, S, V ),
      not equal ( V, nil ),
      instantiate-body ( G1, S, G2 ),
      complete-parlog-semantics ( P, G2, F1 ),
      suspended( F1, V1 ).

suspended-clause-on-var( P, F, A, M, cl( H, G, _ ), V )
    ⇐ L-rename ( F, cl( H, G, _ ), cl( H1, G1, _ ) ),
      L-input-unify ( A, H1, M, S, nil ),
      instantiate-body ( G1, S, G2 ),
      complete-parlog-semantics ( P, G2, F1 ),
      suspended ( F1, V ).
```

is-not-a-suspended-clause#5:
is-not-a-suspended-clause(P, F, A, M, C) then if P is a program, F a partial and/or tree, A an atom and M the associated mode declaration then C is not a suspended clause for the resolution of A in the program P and in the context of F.

```
is-not-a-suspended-clause( P, F, A, M, cl( H, G, _ ) )
    ⇐ L-rename ( F, cl( H, G, B ), cl( H1, G1, _ ) ),
      L-input-unify ( A, H1, M, S, nil ),
      instantiate-body ( G1, S, G2),
      complete-parlog-semantics ( P, G2, F1 ),
      not    is-a-suspended-tree( F1 ).

is-not-a-suspended-clause( P, F, A, M, cl( H, G, _ ) )
    ⇐ L-rename ( F, cl( H, G, B ), cl( H1, G1, _ ) ),
      L-input-unify ( A, H1, M, S, V ),
      not equal ( V, nil ),
      instantiate-body ( G1, S, G2),
      complete-parlog-semantics ( P, G2, F1 ),
      is-a-failure-tree( F1 ).

is-not-a-suspended-clause( P, F, A, M, cl( H, _ , _ ) )
    ⇐ L-rename ( F, cl( H, _ , _ ), cl( H1, _ , _ )),
      not input-unifiable( A, H1 ).
```

instantiation#3 :
instantiation(F, S, F1) then if F is a forest and S a substitution then F1 is the forest obtained after instantiation with S of the nodes of F.

```
instantiation( vid, _ , vid ).

instantiation( for( N, F1, F2 ), S, for( M, F3, F4 ) )
    ⇐ instantiate-node( N, S, M ),
      instantiation( F1, S , F3 ),
      instantiation( F2, S, F4 ).
```

instantiate-node #3 :
instantiate-node(N, S, M) then if N is a node and S a substitution then M is the node obtained after instantiation with S .

```
instantiate-node( N, S, M )
    ⇐ node N is nd( I, A, V, W ),
      L-update( V, S, V1 ),
      L-instance( A, S, A1 ),
      node M is nd( I, A1, V1, W ).
```

buildtree#6 :
buildtree(P, F, N, C, S, F1) then if P is a program,F a partial and/or tree , N an active leaf of F, C a candidate clause for the goal carried by N then F1 is the forest reduced to the forest associated to the guard and whose the other roots are the sons of N, S being the MGU of the goal carried by N and of the head of the clause C.

```
buildtree( P, F, nd( I, A, nil, no-wait ), cl( H, G, B ), S,F1)
    ⇐ L-rename( F, cl( H, G, B ), cl( H1, G1, B1 )),
      L-input-unify( A, H1, M, S1, nil ),
      instantiate-body( G1, S1, G2 ),
      complete-parlog-semantics( P, G2, F2 ),
      L-subs( F2, S2 ),
      L-unify( A, H1, S ),
      instantiate-body( B1, S, B2 ),
      instantiate-body( B2, S2, B3 ),
      build-list-of-node( B3, L ),
      build-root( 0.I, L, F3),
      appendforest( F2, F3, F1 ).
```

possible-tree#4:
possible-tree(P, F, N, C) then if P is a program, F a and/or tree, N an active leaf and C a candidate clause for the resolution of the goal carried by N then it is possible to build sons for the node N.
not possible-tree(P, F, N, C) then it is not possible to build sons for the node N.

```
possible-tree( P, F, N, C )
    ⇐ buildtree( P, F, N, C, _ , _ ).
```

build-list-of-node#2:
build-list-of-node(B, L) then if B is a goal then L is the list of nodes carriing the atoms of goal B, and having the same dewey-number 0.

```
build-list-of-node( A, nd( 0, A, nil, no-wait ).nil )
    ⇐ L-literal( A ).

build-list-of-node( fork( B, A ), L )
    ⇐ L-literal( A ),
      build-list-of-node( B, L1 ),
      append( L1, nd( 0, A, nil, no-wait ).nil , L ).

build-list-of-node( and( B, A ), L )
    ⇐ L-literal( A ),
      build-list-of-node( B, L1 ),
      append( L1.nd( 0, A, nil, no-wait ).nil, L ).

build-list-of-node( fork( B, A ), L )
    ⇐ not L-literal( A ),
      build-list-of-node( B, L1 ),
      build-list-of-node( A, L2 ),
      append( L1, L2, L ).

build-list-of-node( and( B, A ), L )
    ⇐ not L-literal( A ),
      build-list-of-node( B, L1 ),
      build-list-of-node( A, L2 ),
      append( L1, L2, L ).
```

build-root#3:
build-root(I, L, F) then if I is a Dewey number and L a list of nodes then F is the forest whose roots are the nodes of L numeroted from I.

```
build-root( I , nil, vid ).

build-root(J.I,nd(_, A, V, W).L,for( nd( J.I,A,V,W),vid, F))
    ⇐ build-root( s(J).I, L, F ).
```

suspension#4 :
suspension(F, N, V, F1) then if F is a partial and/or tree , N a node of F, V a list of variables then F1 is the tree obtained from F by suspending N on the set of variables V.

```
suspension( for( N, F1, F2 ), N, V, for( N1, F1, F2 ) )
    ⇐ node N is nd( I, A, _ , W ),
      node N1 is nd( I, A, V, W ).

suspension( for( N, F1, F2 ), M, V, for( N, F3, F2 ) )
    ⇐ suspension( F1, M, V, F3 ).

suspension( for( N, F1, F2 ), M, V, for( N, F1, F3) )
    ⇐ suspension( F2, M, V, F3 ).
```

complete-parlog-semantics#3 :
complete-parlog-semantics(P, G, F) then if P is a program, G a goal then F is a complete associated and/or tree.

```
complete-parlog-semantics( P, G, F )
    ⇐ node N is nd( nil, root, nil, no-wait ),
      build-complete-and/or-tree
      (pred(cl(root,true,G ),nil).P,for( N, vid, vid ), F ).
```

build-complete-and/or-tree#3 :
build-complete-and/or-tree(P, F, F1) then if P is a program, F a partial and/or tree then F1 is the complete and/or tree obtained from F.

```
build-complete-and/or-tree( _ , F, F )
    ⇐ is-a-success-tree( F ).

build-complete-and/or-tree( _ , F, F )
    ⇐ is-a-suspended-tree ( F ).

build-complete-and/or-tree( _ , F, F )
    ⇐ is-a-failure-tree ( F ).
```

```
build-complete-and/or-tree( P, F, F1 )
    ⇐ not is-a-failure-tree ( F ),
      active-leaves ( F , L ),
      L-first-active-leaf ( L, N ),
      treatment( P, F, N, F2 ),
      build-complete-and/or-tree( P, F2, F1 ).
```

UTILITIES.

leaves#2 :
leaves (F, L) then if F is a and/or tree then L is the list of leaves of F.

```
leaves (vid, nil ).
leaves (for( N , vid, F2 ), N.L )
    ⇐ leaves ( F2, L ).
leaves (for( N , F1, F2 ), L )
    ⇐ leaves ( F1, L1),
      leaves ( F2, L2),
      append( L1, L2, L ).
```

last-leaf#2 :
last-leaf (F, N) then if F is a and/or tree then N is the rigthmost position leaf of F .

```
last-leaf (for(N, vid,vid), N ).
last-leaf (for( N, F1, F2 ), M ) ⇐ last-leaf ( F2, M ).
```

goal#2 :
goal(N , A) then if N is a node then A is the atom carried by N.

```
goal( nd( _ , A , _ , _ ), A ).
```

nodenumber#2 :
nodenumber(N , I) then if N is a node then I is the number of N.

```
nodenumber( nd( I, _ , _ , _ ), I ).
```

append#3 :
append(L1, L2, L3) then if L1 and L2 are lists then L3 is the list concatenation of L1 and L2.

```
append ( nil, L, L ).
append ( X.L1, L2, X.L3 ) ⇐ append ( L1, L2 , L3 ).
```

member#2 :
member(X, L) then if L is a list then X is an element of L.

```
member ( X, X.L ).
member ( X, Y.L ) ⇐ member ( X, L ).
```

equal#2 :
equal(X , Y) if and only if X is equal to Y.

```
equal ( X, X ).
```

addforest#4 :
addforest(F, N, T, F1) then if F is a forest , N a leaf of F, T a forest then F1 is the forest obtained from F by rooting the forest T in N.

```
addforest( for( N, vid, F2), N, T, for( N, T, F2 ) ).

addforest( for( N, F1, F2), M, T, for( N, F3, F2 ) )
    ⇐ addforest( F1, M, T, F3 ).

addforest( for( N, F1, F2), M, T, for( N, F1, F3 ) )
    ⇐ addforest( F2, M, T, F3 ).
```

preceding-tree#3 :
preceding-tree(F, N, T) then if F is a forest , N a leaf of F which is not the first one, then T is the tree whose root is the sibling preceding N in F.

```
preceding-tree( for( _ , F1, _ ), N, T )
        ⇐   preceding-tree( F1, N, T ).

preceding-tree( for( M, F1, for( N, vid, _ )), N, for( M,
                                                F1, vid ) ).

preceding-tree( for( _ , _ , for( _ , F1, _ ) ), N, T )
        ⇐   preceding-tree( F1, N, T ).

preceding-tree( for( _ , _ , for( M, F1, F2 )), N, T )
        ⇐   preceding-tree( for( M, F1,F2 ), N, T ).
```

addchild#4 :
addchild(F, N, M, F1) then if F is a tree, N a leaf of F, M a node then F1 is the tree obtained from F by adding M as an offspring of N.

```
addchild( F, N, M, F1 )
        ⇐   addforest( F, N, for( M, vid, vid ), F1 ).
```

ready#3 :
ready(F, N, F1) then if F is a forest and N a waiting node in F, then F1 is the forest equal to F except that the corresponding node of N is not in wait state.

```
ready( for( N, F1, F2), N, for( N1, F1, F2 ) )
        ⇐   node N is nd( I, A, V, wait ),
            node N1 is nd( I, A, V, no-wait ).

ready( for( M, F1, F2), N, for( M, F3, F2 ) )
        ⇐   ready( F1, N, F3 ).

ready( for( M, F1, F2), N, for( M, F1, F3 ) )
        ⇐   ready( F2, N, F3 ).
```

appendforest#3 :
appendforest(F1, F2, F3) then if F1 and F2 are forests then F3 is the concatenation of F1 and F2.

```
appendforest( for( N, F1, vid), F2, for( N, F1, F2 ) ).

appendforest( for( N, F1, F2), F3, for( N, F1, F4 ) )
        ⇐   appendforest( F2, F3, F4 ).
```

RELATIVE DENOTATION

L-unify#5 :	L-unify(A, B, M, S, V) for all literals A and B, all mode declarations M, all substitutions S such that S is a MGU of the lists of input terms of A and B, and V is the set of suspended variables .(eventualy, V is empty)
L-unify#3 :	L-unify(A, B, S) for all literals A and B, and all substitutions S such that S is a MGU of A and B.
L-first-active-leaf#3:	L-first-active-leaf(L, N) for all the lists of active nodes and every node N such that N is the 1^{rst} node to execute the commit operator or suspends
L-instance#3:	L-instance(T, S, T1) for all terms T and T1, all substitution S such that T1 is the instantiation of T by S.
L-literal#1 :	L-literal(A) for the representation of all litterals in the PARLOG dialect.
L-mode#2 :	L-mode(Q, M) for all packet representations Q of PARLOG and all representations of possible associated modes.
L-packet#4 :	L-packet(P, A, M, Q) for all literals A, all programs P and Q , all mode declarations M such that Q is the packet of clauses corresponding to A in program P and M is the associated mode declaration .
L-random-choice#2 :	L-random-choice(S, C) for all sets of clauses S and clauses C such that C is a random choice from S .
L-rename#3 :	L-rename(F, T, T1) for every forest F, all terms T and T1 such that T1 is a renaming of T with variables not appearing in F.
L-same-predicate#2 :	L-same-name(P, C) for all clauses C and all packets P such that clause C defines the same predicate than P.
L-subs#2 :	L-subs(F, S) for all success **and/or** trees F and substitutions S such that S is the answer substitution induce by F.
L-substitution#1 :	L-substitution(S) for all substitutions S.
L-update#3 :	L-update(L, S, L1) for all sets of variables L and L1, and substitutions S such that L1 is composed of the sets resulting from updating elements of L relative to S if none of these results is empty otherwise L1 is empty. A set of variables V1 is the result of updating a set of variables V relative to a substitution S if V1 is composed of elements of V which are not in the domain of S. V1 is empty if S assigns all variables in V.
L-variable#1 :	L-variable(V) for all variables V of the PARLOG dialect.

Notes :

The predicates **L-packet** and **L-same-name** are ifmply defined when we choose a representation for literals, which is not necessary.

The predicate **L-random-choice** which expresses that its second argument is an element chosen at random from its first argument, is in fact dependent on the interpretation of 'random' by the implementation. In the case of a fair interpreter, this predicate specifies simply the inclusion test of an element in a set.

The predicates **L-first-active-leaf** is also implementation dependent. In the case of a fair interpreter, this predicate also simply specifies the inclusion test of an element in a set.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique