Semantique de PARLOG, un langage logique parallele

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° 815

SEMANTIQUE DE PARLOG
UN LANGAGE LOGIQUE PARALLELE

Gilles RICHARD
Antoine RIZK

MARS 1988

Sémantique de PARLOG, un langage logique parallèle

G. RICHARD - A. 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-PS210). 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 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

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 #951 COCOS.

Introduction.

L'écriture d'un programme adapté à une exécution parallèle est en général, une tâche assez difficile. De ce fait, le choix de langages pour la programmation parallèle est un problème crucial : la sémantique doit être claire et le parallélisme inhérent à leur structure. Les langages de programmation logique et, en particulier les clauses de Horn, possèdent ces qualités. En effet, étant donné un ensemble de clauses de Horn, il existe un grand nombre de stratégies de résolution différentes de celle adoptée dans PROLOG Standard, et notamment des stratégies parallèles Ces stratégies ont donné naissance à des langages qui, tout en augmentant la puissance d'expression des clauses de Horn, ne conservent pas, en général, une sémantique purement déclarative. PARLOG est l'un de ces langages et sa sémantique opérationnelle était simplement décrite en langage naturel par ses concepteurs [CG 86].

Nous présentons donc ici une spécification formelle de la sémantique d'un interpréteur PARLOG, en utilisant une méthode déjà expérimentée pour un langage séquentiel déterministe (Prolog standard [DF 87b], [DR 87]) mais inédite pour un langage concurrent non déterministe.

Remarquons qu'il ne s'agit pas de spécifier l'ensemble des prédicats prédéfinis de PARLOG, mais seulement de définir une sémantique pour les différentes facilités de contrôle autorisées en PARLOG (déclaration de modes, opérateur de recherche séquentielle, opérateur de conjonction séquentielle, opérateur d'engagement). En ce sens, notre travail constitue plus une étude de faisabilité qu'une description exhaustive des particularités du langage.

1) le langage PARLOG

Intrinsèquement à la résolution qui gouverne l'exécution des programmes logiques [ROB 65], il y a deux sources principales de non-déterminisme, qui sont autant de potentialités pour l'interprétation concurrente. Premièrement, dans la sélection de l'atome dans le but et deuxièmement dans la sélection de la clause. Les deux formes de parallélisme qui en résultent sont nommées respectivement parallélisme **et** et parallélisme **ou**.

PARLOG est le dernier-né d'une famille de langages similaires qui dérivent de Relationnal Language (Clark et Gregory 1981) et comprend Concurrent Prolog [Sha 86] et Guarded Horn Clauses [Ued 85b].

PARLOG (Clark et Gregory 1984) est un langage de programmation concurrente qui met en œuvre le parallélisme **et** par l'intermédiaire de la communication de flot inter-processus et le parallélisme **ou** par l'usage de clauses de Horn gardées et du non déterminisme **don't care** .

a) syntaxe d'un programme PARLOG

Nous donnons ici une syntaxe simplifiée suffisante dans un premier temps pour ce que nous voulons réaliser.

Un programme PARLOG est constitué par :

- un ensemble fini de paquets : un paquet définissant un prédicat est un ensemble fini de clauses de Horn gardées, chaque clause étant séparée de la clause suivante par l'**opérateur de recherche parallèle** noté '.' ou l'**opérateur de recherche séquentielle** noté ';'

- un ensemble de déclarations de modes associées à chaque paquet

Une clause de Horn gardée est de la forme :

$$H \text{ <- } G_1 \text{ op } ... \text{ op } G_n : B_1 \text{ op } ... \text{ op } B_m.$$

où H est un atome (tête de clause), $G_1,...,G_n$ un ensemble d'atomes qui constitue la garde et $B_1,...,B_m$ un ensemble d'atomes qui constitue le corps de la clause.

Le symbole ':' qui sépare la garde du corps dénote l'**opérateur d'engagement**.

Le symbole op dénote l'un des deux opérateurs suivants : **',', conjonction parallèle** ou bien **'&', conjonction séquentielle.**

La garde existe toujours mais elle peut éventuellement être réduite à **true** (que l'on n'est pas alors obligé d'écrire).

Une déclaration de modes associée à la définition du prédicat P d'arité n est de la forme :

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

où m_i est l'un des deux symboles input ou output dénotés **?** et **^** en syntaxe concrète.

Remarque : La déclaration de modes est associée au symbole de prédicat et non à la clause comme en Concurrent Prolog. Il n'y a qu'une seule déclaration de modes par prédicat.

b) sémantique déclarative

La sémantique logique d'une clause gardée est celle usuelle d'une clause de Horn (sans garde):

$H <- G_1,...,G_n ,B_1,...,B_m$. où la garde fait partie intégrante du corps de clause.

Si G_1 et.... et G_n et B_1 et ... et B_m sont vrais alors H est vrai.

On voit donc qu'à l'intérieur d'une clause, la lecture déclarative de l'opérateur d'engagement est celle d'une conjonction. Cependant, dès lors qu'il existe plusieurs clauses pour définir le même prédicat, une simple lecture déclarative ne suffit pas à rendre compte des résultats donnés par l'interpréteur : une lecture opérationnelle devient indispensable.

c) sémantique opérationnelle

La sémantique opérationnelle est tout à fait différente de celle bien connue de l'interpréteur PROLOG Standard. C'est seulement à ce niveau qu'intervient le non déterminisme et la spécificité du langage PARLOG.

Donnons d'abord quelques définitions :

Soit t un terme : on note V(t), l'ensemble des variables apparaissant dans t.

Soit s une substitution et V,un ensemble de variables : on note s/V la restriction de s à V, c'est à dire la substitution définie par :

$$(s/V)(x) = s(x) \text{ si } x \in V, \text{ sinon } (s/V)(x) = x.$$

<u>Déf 1</u>: <u>contrainte d'entrée</u>

Soient $A = p(a_1,\cdots, a_n)$, un atome et $H = p(t_1,\cdots, t_n)$, une tête de clause renommée avec des variables n'apparaissant pas dans A. Notons mode $p(m_1,\cdots, m_n)$, la déclaration de modes associée au prédicat p.
Supposons que m_i soit un mode d'entrée et soit s_i un unificateur de a_i et t_i : on dit que s_i satisfait la **contrainte d'entrée** si et seulement si :

$$s_i/V(a_i) \text{ est la substitution vide .}$$

En d'autres termes, s_i n'instancie pas les variables apparaissant dans a_i. Si nous convenons de noter dom(s), le domaine d'une substitution s, la condition précédente est équivalente à :

$$dom(s_i) \cap V(a_i) = \varnothing.$$

Déf 2: variables de suspension

Si s_i ne satisfait pas la contrainte d'entrée, cela signifie que : $dom(s_i) \cap V(a_i) \neq \varnothing$.

Toute variable de l'ensemble $dom(s_i) \cap V(a_i)$ est appelée **variable de suspension.** L'ensemble $\cup(dom(s_i) \cap V(a_i))$ pour a_i décrivant l'ensemble des arguments spécifiés en entrée est l'ensemble des variables de suspension.

Déf 3: clause candidate

Soit $A = p(t_1,..., t_n)$, un atome : une clause définissant le prédicat p est une **clause candidate** à la résolution de A si et seulement si :

- la liste des termes d'entrée dans la tête de clause après renommage s'unifie avec la liste des termes d'entrée dans A
- l'ensemble des variables de suspension est vide
- la garde de la clause instanciée par l'unificateur précédent admet une substitution réponse

Dans le cas où la garde est réduite à **true**, les deux premières conditions sont suffisantes pour que la clause correspondante soit candidate.

Remarquons qu'il n'y a aucune contrainte sur les arguments spécifiés en sortie dans la déclaration de modes.

Exemples :

mode p(?,?,^)

A = p(f(X), X, Y)

H = p(f(1), Z, Z)

L'unificateur de la liste des termes d'entrée est : $X \leftarrow 1, Z \leftarrow 1$.

La contrainte d'entrée n'est pas satisfaite par s1 : $X \leftarrow 1$. X est une variable de suspension : une telle unification sera suspendue par l'interpréteur PARLOG sur X.

A = p(f(X), X, 1)

H = p(f(1), Z, 2) : suspension sur X pour les mêmes raisons que précédemment.

A = p(f(X), X, 1)

H = p(f(Y), Z, Z)

L'unificateur de la liste des termes d'entrée est : $Y \leftarrow X, Z \leftarrow X$.

s1 vaut : $Y \leftarrow X$ et s2 vaut : $Z \leftarrow X$: donc il n'y a pas de variables de suspension : cette unification n'est pas suspendue.

A = p(f(X), 1, Y)

H = p(f(Z), Z, T)

L'unificateur de la liste des termes d'entrée est : $X \leftarrow 1, Z \leftarrow 1$

s1 vaut : $Z \leftarrow X$ et s2 vaut : $Z \leftarrow 1$: donc il n'y a pas de variables de suspension : cette unification n'est pas suspendue.

Le fonctionnement d'un interpréteur PARLOG peut alors se décrire globalement de la façon suivante :

- étant donné un but B, la résolution des atomes de B séparés par la conjonction parallèle s'effectue en parallèle (un processus étant créé pour la résolution de chaque atome de B), celle d' atomes séparés par la conjonction séquentielle s'effectue séquentiellement.

- étant donné un atome A, la recherche des clauses candidates à la résolution de A s'effectue en parallèle pour les clauses séparées par l'opérateur de recherche parallèle, séquentiellement pour les clauses séparées par l'opérateur de recherche séquentielle.

- parmi toute les clauses candidates à la résolution de A, une et une seule est sélectionnée pour poursuivre la résolution. La sélection se fait **au hasard** s'il n'y a pas d'opérateur de recherche séquentielle séparant ces clauses ; sinon, le choix se fait en respectant l'ordre indiqué par l'opérateur de séquentialité. L'opérateur d'engagement peut donc être vu comme la version concurrente du cut de PROLOG Standard en ce sens qu'il supprime des choix potentiels : de ce fait , il participera de l'incomplétude de l'interpréteur. Le résultat de l'instanciation des variables est communiqué aux autres atomes (ou processus).

- s'il n'existe pas de clauses candidates à la résolution de l'atome A, mais qu'il existe des clauses potentiellement utilisables ne satisfaisant pas la contrainte d'entrée (clauses suspendues), alors la résolution de A est suspendue (ce n'est pas un échec) jusqu'à ce que A soit suffisamment instancié pour qu'une de ces clauses devienne candidate ou bien qu'il n'y ait plus de clauses suspendues.

Les contraintes de modes ne créent pas d'échecs supplémentaires : elles ne font que suspendre la résolution de certains buts et deviennent le moyen de synchroniser les processus en PARLOG.

Les opérateurs de recherche séquentielle des clauses et de conjonction séquentielle ainsi que les déclarations de modes constituent des primitives de contrôle de la stratégie de l'interpréteur.

d) propriétés de l'interpréteur PARLOG

correction : il est clair qu'un interpréteur PARLOG est correct au sens de [KOW84] : la substitution réponse qu'il produit est une réponse logique (naturellement, l'absence d'occur-check pour un interpréteur PARLOG et l'existence de primitives impures telle que la négation mettent en défaut cette correction).

complétude : il est aussi clair qu'un interpréteur PARLOG est incomplet car :

- le choix de la clause candidate peut générer une boucle infinie qui ne donnera pas de solution alors qu'il peut par ailleurs exister des solutions logiques

- le choix de la clause candidate peut conduire simplement à un échec alors qu'une autre clause aurait produit des solutions

On doit remarquer 5 points qui distinguent fondamentalement PARLOG de l'interpréteur PROLOG standard:

- PARLOG n'utilise pas le backtracking

- un programme PARLOG n'est pas réversible de par l'utilisation des modes : il n'y a pas d'interchangeabilité entre les arguments d'entrée et les arguments de sortie

- PARLOG fournit **au plus** une réponse parmi l'ensemble des réponses possibles

- PARLOG ne donne pas nécessairement deux fois de suite la même réponse au même couple programme-but

- la sémantique opérationnelle d'un programme PARLOG sans conjonction séquentielle et sans opérateur de recherche séquentielle (c'est à dire où l'on ne contrôle pas la stratégie) est indépendante de l'ordre des clauses et de l'ordre des atomes dans les corps de clauses

e) notion de garde sûre

Etant donnée une clause H ← G : B, sa garde G est dite sûre si et seulement si, pour n'importe quel appel A tel que σ soit le MGU de H et A avec renommage éventuel, alors toute substitution réponse s obtenue à partir de σ(G) est telle que s/V(H) soit vide. En d'autres termes, s n'instancie pas les variables de la tête de clause.

Une clause est sûre si sa garde est sûre et un programme est sûr si toutes ses clauses sont sûres.

Tous les programmes PARLOG sont sûrs : on dit parfois que PARLOG est un langage sûr.

L'intérêt de cette restriction est qu'elle évite la manipulation d'environnements supplémentaires, nécessaires au stockage des liaisons multiples avant l'exécution de l'opérateur d'engagement.

f) exemples

Exemple 1 : considérons le programme suivant qui spécifie l'appartenance d'un élément à une liste :

member(x, [x|l]).

member(x, [y|l]) <- member(x, l).

Un interpréteur Prolog donnera trois réponses positives à la question :

member(1, [1,9,1,2,3,1])

Un interpréteur PARLOG, avec la déclaration de mode suivante : mode member(?, ?), ne donnera naturellement qu'une seule réponse positive : en effet, l'opérateur d'engagement implicite sélectionnera une seule clause candidate parmi les deux clauses du programme et supprimera tout backtracking. En ce sens, l'effet de l'engagement est équivalent à celui d'un cut Prolog à la fin de la première clause.

Exemple 2 : considérons maintenant le programme suivant qui spécifie que la différence de ses deux arguments est une liste différentielle :

dlist(x, x).

dlist([x|l1], l2)) <- dlist(l1, l2).

Si on pose à l'interpréteur Prolog la question :

dlist([1|x], x)

on va obtenir une réponse positive. Cependant, l'effet du backtrack sera d'amener l'interpréteur à parcourir une branche infinie de l'arbre de recherche associé, dûe à l'existence de variables insuffisamment instanciées.

En PARLOG, une déclaration de modes du type mode dlist(?, ^) suffit à résoudre ce problème : en effet, la contrainte de modes fait en sorte que seule la clause souhaitée est candidate à la résolution, supprimant de ce fait la boucle infinie.

Exemple 3 : soit le programme classique qui partitionne une liste en deux parties, l'une constituée des éléments supérieurs à un élément donné, l'autre constituée des éléments inférieurs ou égaux à cet élément.

partition(a, [x|l],[x|l1], l2) <- x>a : partition(a, l, l1, l2).

partition(a, [x|l], l1, [x|l2]) <- x=<a : partition(a, l, l1, l2).

partition(a, [], [], []).

Remarquons qu'il est indispensable d'introduire une garde dans la deuxième clause car la stratégie de recherche parallèle en PARLOG pour les clauses candidates ne garantie pas que le test x>a sera exécuté : cela oblige à une écriture symétrique des deux clauses récursives et l'ordre des clauses est sans importance. Au contraire, la stratégie Prolog standard permet l'écriture de la deuxième clause sans le test en utilisant un cut dans la première clause. Dans ce cas, PARLOG oblige à une écriture plus déclarative.

2) l'arbre et/ou, objet sémantique

a) l'arbre et/ou

L'arbre **et/ou** associé à un couple programme-but est un moyen de représenter la résolution des problèmes (Kow 84). D'autre part, cet arbre est bien adapté à la représentation formelle de l'exécution parallèle des clauses de Horn pour les raisons suivantes :

- premièrement, l'arbre **et/ou** permet d'exhiber explicitement les différents constituants d'un but qui peuvent effectivement être traités en parallèle par l'interpréteur (parallélisme **et**)

- deuxièmement, les différentes branches **ou** correspondant aux différentes clauses utilisables pour la résolution d'un but exhibent les possibilités de parallélisme **ou**

Etant donnés un programme P et un but G, on peut leur associer un unique arbre **et/ou** défini de la façon suivante:(on peut toujours se ramener au cas où G est un atome en ajoutant au programme l'unique clause : question <- G.)

- chaque nœud porte un atome.

- si le nœud est un nœud **ou** portant l'atome A , il a autant de fils que de clauses dans P dont la tête s'unifie avec l'atome A (éventuellement aucun s'il n'existe pas de telles clauses). Chaque fils est un nœud **et**.

- si le nœud est un nœud **ou** portant l'atome A, alors pour chaque clause de P

  A0 <- A1,...,An.

  dont la tête s'unifie avec A, le MGU étant s, alors il y a un fils **et** qui porte l'atome s(A0) et ce nœud possède n fils **ou** qui portent respectivement les atomes s(A1),..., s(An). Si n = 0, (cas d'un fait), le nœud **et** est une feuille.

- enfin, le nœud racine est un nœud **ou**

<u>algorithme de construction de l'arbre et/ou pour un programme et un but donnés</u>

G : but

P : programme

T : arbre et/ou

P : = P ∪ { question <-G.}

T : = (question, or)

tant qu'il existe une feuille (A, or) pour laquelle il existe dans P une tête de clause renommée qui s'unifie avec a, faire :

pour chaque clause renommée A0 <- A1,..., An dont la tête s'unifie, faire :

s : = MGU(A, A0) ;

T : = T{(A, or) <- (A, or)
‖
(s(A), and)
/ \
(s(A1), or) ... (s(An), or) }

La construction s'arrête lorsqu'il n'existe plus de feuille **ou** ou bien lorsque les feuilles **ou** restantes n'ont pas de clauses correspondantes.

b) exemple

Traitons un exemple : (variables en minuscules, constantes en majuscules)

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, S).

goal3(B, Q).

goal4(R, S).

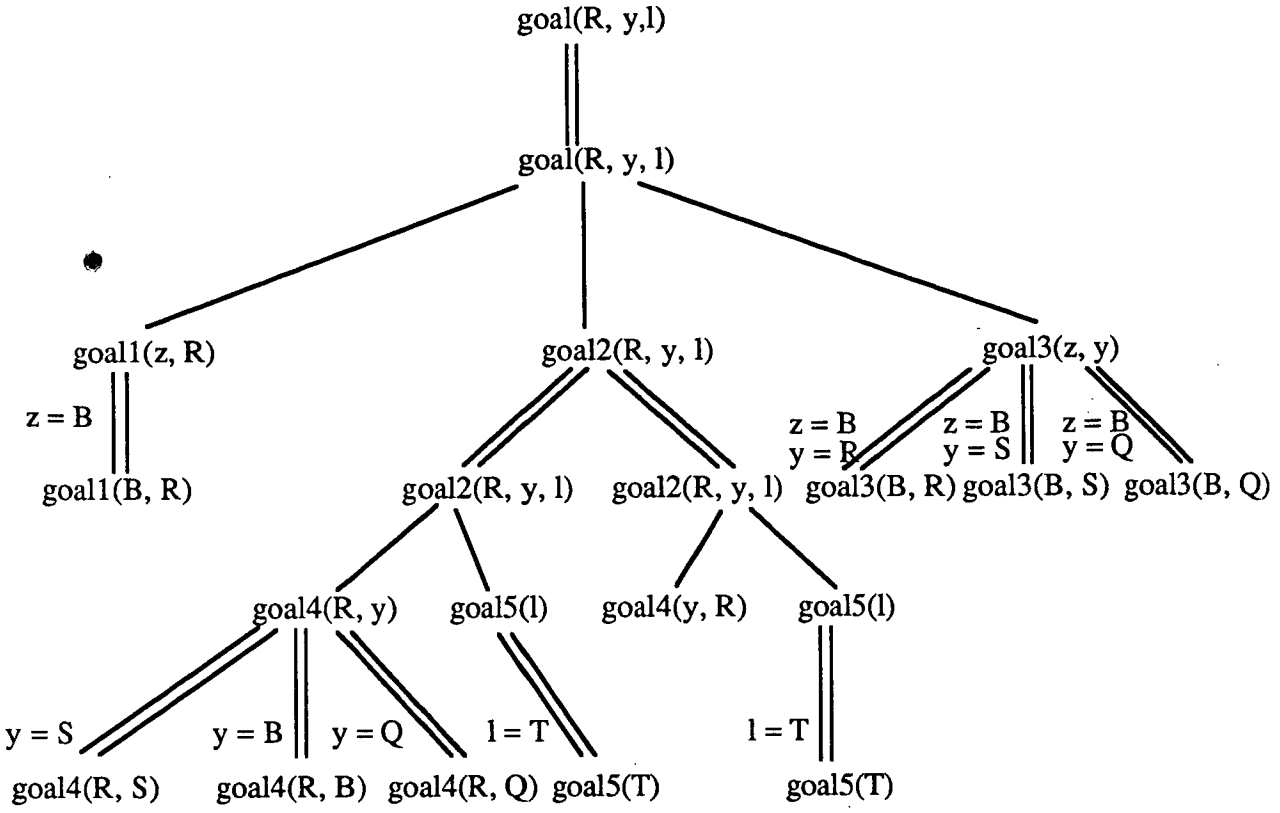
goal4(R, B).

goal4(R, Q).

goal5(T).

Par convention, les branches issues d'un noeud ou seront notées // et les branches issues d'un noeud et seront notées /.

L'arbre **et/ou** correspondant à la question goal(R,y,l) est le suivant :



Les substitutions sont associées aux branches correspondantes. Les substitutions réponses sont obtenues en composant entre elles les substitutions compatibles. On voit ici les deux substitutions réponses : y = S, l = T et y = Q, l = T.

c) la sémantique de PARLOG en termes d'arbre et/ou

On peut modéliser la sémantique d'un interpréteur PARLOG auquel on soumet un couple programme-but comme la construction d'un sous arbre instancié de l'arbre **et/ou** précédent : en effet, la sélection d'une seule clause parmi les clauses candidates se modélise par le fait que chaque nœud **ou** possède au plus un nœud fils, le reste étant inchangé : l'effet de l'opérateur d'engagement s'interprète comme la suppression de toutes les branches **ou** sauf une, issues d'un même atome .

D'autre part, la communication aux autres atomes (processus) des résultats de l'unification d'un atome avec une tête de clause candidate se modélise par le fait que ces atomes sont instanciés par l'unificateur.

Il convient de remarquer que ce sous arbre n'est pas unique : en effet, la sélection d'une autre clause candidate à un moment donné de l'exécution conduit à la construction d'un sous arbre différent. Cela correspond au fait que le même interpréteur PARLOG ne donne pas nécessairement la même réponse au même couple programme-but à des instants différents. Cependant, si un sous arbre est potentiellement constructible par l'interpréteur à partir d'un couple programme-but donné, et si l'interpréteur est **équitable**, alors ce sous arbre sera construit si on soumet un certain nombre de fois le même couple programme-but à l' interpréteur. Donc, **l'ensemble des sous arbres potentiellement constructibles** constitue en fait la sémantique d'un couple programme-but modulo la stratégie de l'interpréteur PARLOG. Remarquons que ceci signifie que nous distinguerons deux exécutions d'échec.

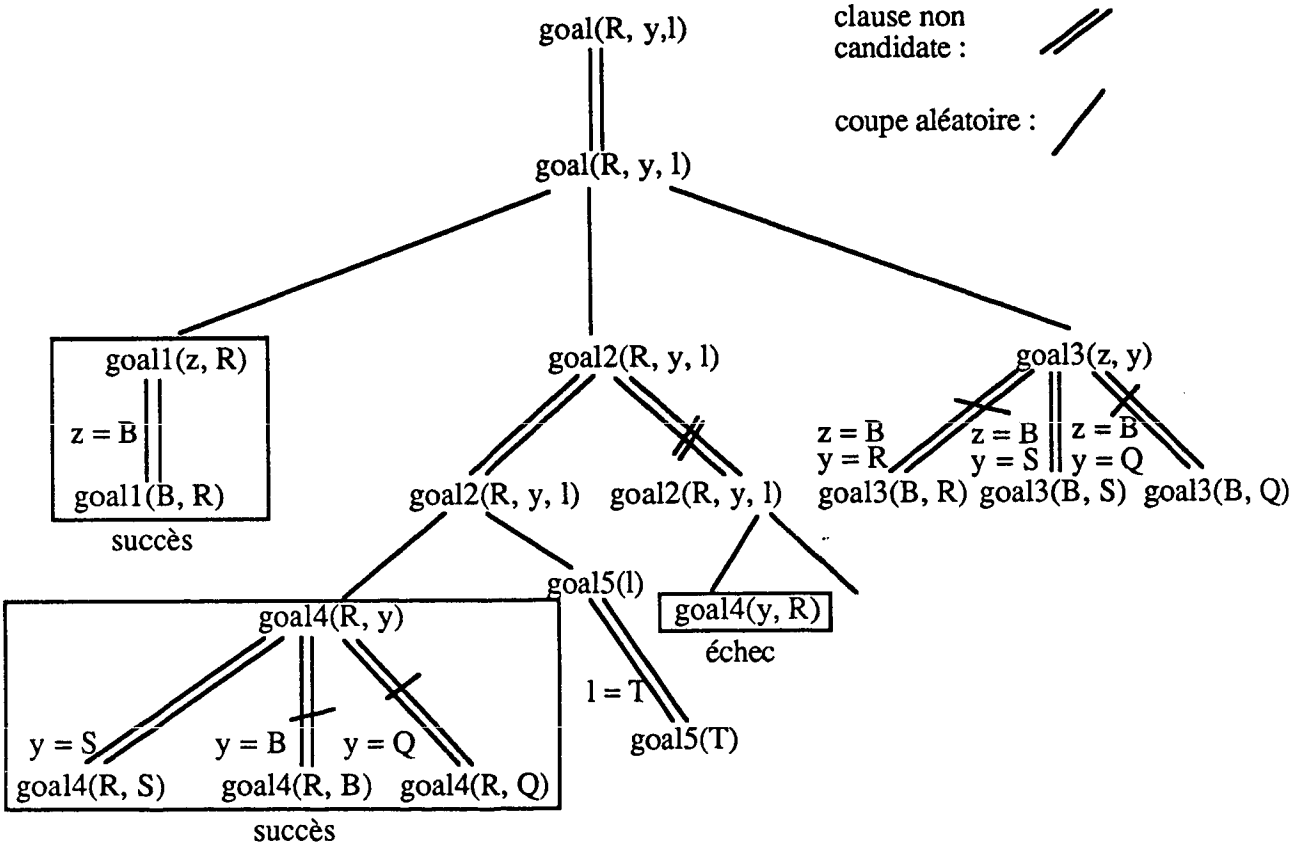
Traitons l'exemple précédent en PARLOG :

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).

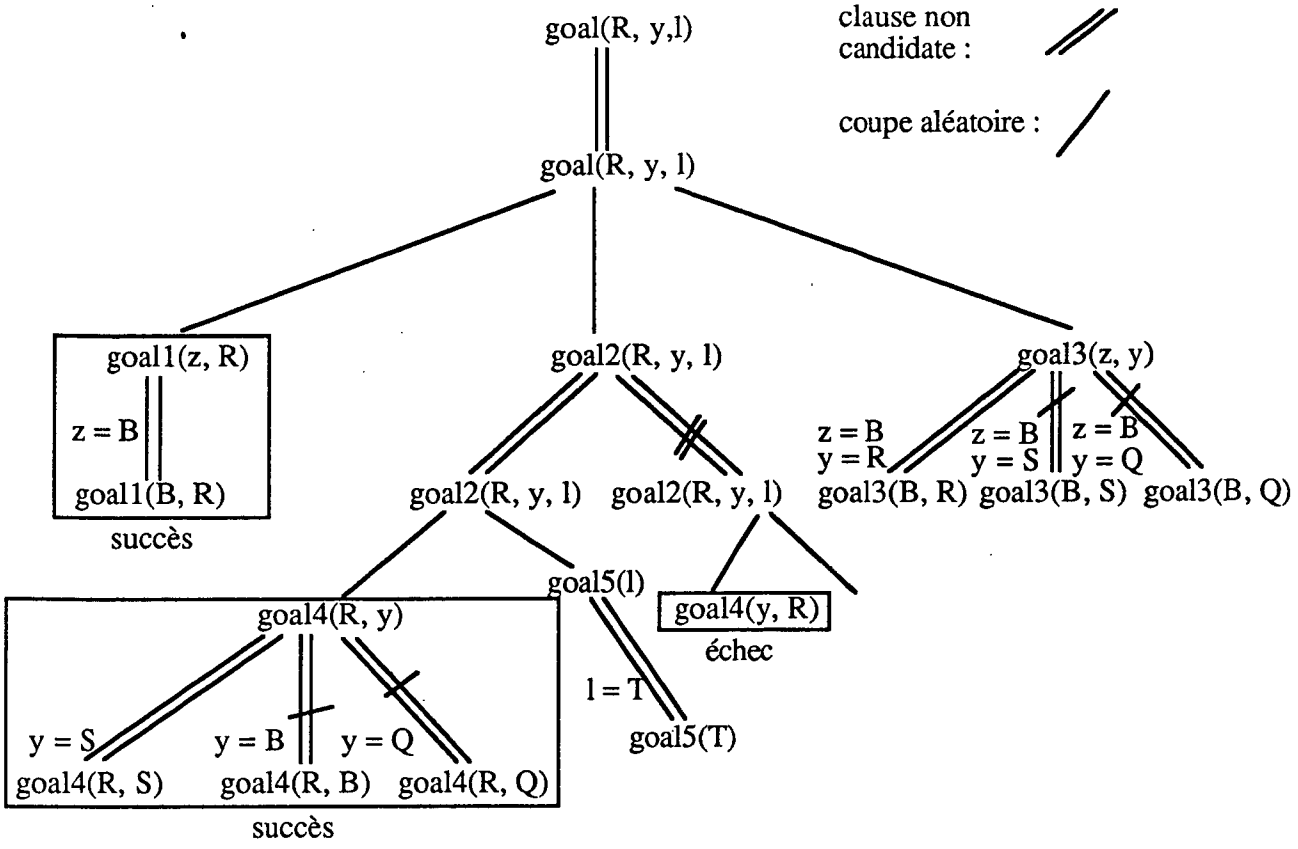
Les représentations des gardes sont encadrées.

Un arbre serait :



La substitution fournie par l'interpréteur PARLOG dans ce cas est : y = S, l = T.

Une autre exécution pourrait produire l'arbre suivant où le choix de la clause candidate pour goal3 est différent :



On voit que dans ce cas l'interpréteur PARLOG ne fournit aucune réponse (pas de substitution compatible dans l'arbre)

Donc, nous décrirons la sémantique d'un couple programme-but (P,G) PARLOG soumis à l'interpréteur PARLOG comme une relation (il s'agit bien d'une relation et non d'une fonction) liant le programme et le but à un sous arbre **et/ou** associé T : **parlog-semantics**(P, G, T).

4) la spécification formelle

Deux principes essentiels nous guident :

- la lisibilité de la spécification : ceci peut être réalisé par le choix d'un langage de spécification avec une sémantique claire et par la facilité de la réalisation informelle des preuves

- la minimalité de la spécification : la spécification doit décrire toutes les propriétés intéressantes du concept et rien de plus : ceci est réalisé par l'usage de la notion de dénotation relative

a) le langage de spécification et sa sémantique

Nous souhaitons donc un langage formel mais lisible, et autant que possible modulaire. Les clauses de Horn avec négation dans le corps de clauses, que nous avons auparavant utilisé pour la spécification formelle de PROLOG standard [DR 87], répond à ces exigences. Ce langage possède une sémantique purement déclarative complètement formalisée dans les travaux de [DF 87a] auquel il convient de se reporter pour une description complète.

- le langage utilisé est un langage logique du 1er ordre avec lequel nous écrivons des clauses de Horn avec négation.

- la sémantique de ces clauses est purement déclarative et est basée sur la notion d'arbres de preuve avec négation : on peut se reporter aux travaux de P.DERANSART et G.FERRAND sur ce sujet [DF 87a]. De ce fait, l'ordre des clauses et l'ordre des atomes dans les corps de clause n'ont aucune importance.

- afin de pouvoir considérer cette sémantique, nous écrivons un programme finiment fondé, et plus précisément stratifié au sens de [ABW 86]. Le programme compte 4 niveaux de stratification.

b) les méthodes de preuves

La spécification sera d'autant plus compréhensible qu'il sera possible de se convaincre simplement (de manière informelle au moins) de deux propriétés :

1) les transformations d'arbres qui sont décrites sont bien celles attendues : c'est un problème de correction.

2) tous les cas possibles sont bien envisagés : c'est un problème de complétude.

Des méthodes de preuves de correction et complétude ont été développées dans [Der 87] et [DF 87a].

Leur mise en œuvre nécessite :

- d'associer à chaque prédicat sémantique une **assertion de correction** et une **assertion de complétude**.Dans tous les cas , si P est le prédicat considéré et X, Y une partition de l'ensemble de ses arguments, l'assertion de correction sera du type : **si** A(X) **alors** B(X, Y) et l'assertion de complétude du type: A(X) **et** B(X, Y).

- Dans le cas d'un usage négatif de P, l'assertion de correction est du type **si** A(X) **alors** non B(X, Y) et celle de complétude A(X) **et** non B(X, Y).

Cependant, du fait que nous ne disposons pas d'une "spécification" de notre spécification, ces assertions sont écrites en langage naturel et elles doivent être considérées comme des commentaires.

Ces assertions permettent de comprendre comment sont construites les clauses : en effet, en supposant la validité des assertions de correction pour les atomes du corps de clause, on montre que l'assertion relative à l'atome de tête est valide.

Remarquons que l'écriture de la spécification est plus guidée par des impératifs de simplicité des preuves que par un souci d'élégance (on notera des redondances qui, de notre point de vue, facilitent la lecture).

c) la dénotation relative

Soit E un ensemble (éventuellement infini) d'atomes : on peut lui associer le programme suivant E'= { A<-. / A in E }. Alors, la dénotation d'un programme P relativement à E est la dénotation du programme P $\cup$ E'. Le programme E' est le programme le plus simple dont la dénotation est E, en ce sens qu'il ne contient que des faits.

Quand nous désirons avoir dans la dénotation de notre programme un ensemble d'atomes bien définis, alors l'usage de la dénotation relative de P relativement à cet ensemble est un moyen d'éviter plusieurs problèmes:

- d'une part de choisir une représentation sémantique des objets que nous voulons spécifier : c'est le cas des variables PARLOG, des substitutions, des atomes, etc...

- d'autre part d'écrire une description complète de cet ensemble en clauses de Horn : c'est le cas de l'unification, du renommage des clauses, de l'unification directionnelle, etc...

En particulier, l'usage approprié de la dénotation relative permet de ne pas décrire en clauses de Horn la spécification de concepts clairs et indépendants de l'implémentation : une telle description risquerait en effet d'induire des contraintes indésirables sur l'implémentation et ne ferait qu'alourdir la spécification.

5) introduction à la lecture de la spécification

a) structures de données

- pour représenter l'arbre **et/ou**, nous utilisons une structure de forêt : une forêt peut être vue comme une liste d'arbres (un arbre étant un cas particulier de forêt) : cette structure favorise l'écriture récursive des clauses où elle intervient. Une telle structure est représentée à l'aide du symbole de fonction **for** à trois arguments : le premier représente un nœud, les deux autres représentent des forêts.

- pour représenter les nœuds de l'arbre **et/ou**, nous utilisons un symbole de fonction à quatre arguments, **nd** : le premier argument est le numéro du nœud (numérotation à la Dewey), le second argument représente l'atome porté par le nœud, le troisième représente la liste des variables sur lesquelles l'atome est suspendu : une liste vide exprime donc le fait que le processus correspondant n'est pas suspendu. Le quatrième argument est un marqueur qui indique si le nœud est en attente ou non.

- pour représenter une clause, nous utilisons un symbole de fonction à trois arguments, **cl** : le premier argument représente la tête de la clause, le deuxième argument représente la garde et le troisième, le corps.

- un programme est simplement une liste de prédicats (notation pointée des listes).

- un prédicat est un couple paquet-déclaration de modes associée.

- un paquet est une liste de clauses séparées par les opérateurs ',' ou bien ';'.

b) syntaxe du langage de spécification

Le langage de spécification est un langage logique du 1er ordre qui contient :

- ♦ un ensemble dénombrable de variables notées avec une syntaxe Edimbourgh

- ♦ un ensemble fini de symboles de fonctions contenant :
  - arité 0 : true, fail, nil, vid, root, 0,
  - arité 1 : s (pour les naturels)
  - arité 2 : . (pour les listes), fork (et parallèle), and (et séquentiel), seq (recherche séquentielle), par (recherche parallèle), pred
  - arité 3 : for, cl
  - arité 4 : nd

- ♦ un ensemble de symboles de prédicats définis par des clauses (une soixantaine) :
  ex.: **suspended#2**, **addchild#4**, etc...

- un ensemble de 13 symboles de prédicats non définis par des clauses et préfixés par L-: ex. **L-unify#5**, **L-literal#1**, etc...

- trois connecteurs logiques : **not**, $\Leftarrow$, ,

Avec ce langage, nous écrivons des clauses de Horn avec négation.

c) la relation parlog-semantics

L'essentiel de la spécification consiste en la définition de cette relation **parlog-semantics** qui exprime qu'un sous arbre et/ou est dans la sémantique d'un couple programme-but (P, G), c'est à dire est potentiellement constructible par l'interpréteur PARLOG à partir de (P,G).

Cette relation est définie par le biais de la relation **build-and/or-tree** qui spécifie les conditions pour lesquelles un arbre **et/ou** est obtenu à partir de l'arbre **et/ou** de l'étape précédente. A ce niveau, plusieurs cas se présentent : soit l'arbre de l'étape précédente est un arbre succès, soit un arbre suspendu, soit un arbre d'échec auxquels cas la construction s'arrête ; sinon, la construction continue.

La relation **treatment** spécifie alors l'évolution de l'arbre **et/ou** étape par étape, selon que pour l'atome considéré, il existe ou non des clauses candidates.

6) discussion

a) avantages et inconvénients de la méthode

Une spécification formelle est écrite à l'usage des implémenteurs du langage, des utilisateurs du langage et tous ceux qui sont intéressés à une analyse des concepts du langage (équivalence de programmes - propriétés des programmes - etc...).

Notre spécification possède naturellement les propriétés du langage utilisé pour l'écrire : lisibilité, modularité. D'autre part, on peut aisément en dériver une spécification exécutable. Nous en avons fait l'expérience pour Prolog standard où, à partir de la spécification formelle, nous avons déduit une spécification exécutable qui rend en sortie les différents arbres (arbres de recherche partiels dans ce cas).Enfin, cette méthode de spécification permet sans difficulté majeure de traiter des prédicats ou opérateurs prédéfinis dont la sémantique est purement opérationnelle. Pour toutes ces raisons, notre spécification semble bien adaptée aux besoins d'implémenteurs et d'utilisateurs.

Certaines propriétés des programmes s'expriment simplement en terme de sous-arbres **et/ou**. Par exemple, un deadlock potentiel dans l'exécution de P et G s'exprime par le fait qu'il existe un arbre entièrement suspendu dans la sémantique :

potential-deadlock(P, G) $\leftarrow$ **parlog-semantics**(P, G, F),
is-a-suspended-tree (F).

Cependant, le problème des comparaisons de programmes à partir de ces sous-arbres **et/ou** reste ouvert.

b) les autres travaux

Notre spécification, bien qu'utilisant le même langage que Pereira et Monteiro [PM 78] ne décrit pas le fonctionnement de l'interpréteur et en ce sens semble de plus haut niveau.

Ueda [Ued 85] donne un algorithme de résolution, en langage naturel, pour GHC : bien qu'il soit rigoureux et sa compréhension facilitée, les problèmes inhérents à ce type de méthode demeurent (risque d'ambiguïté).

Beckmann [Beck 86], qui utilise CCS (Calculus of Communicating Systems [Mi 80]) et Saraswat [Sar 86] avec une méthode à la Plotkin [Plot 81], construisent des sémantiques entièrement formelles pour des langages logiques concurrents, qui paraissent bien adaptées aux problèmes de preuves d'équivalence et de propriétés de programmes.

c) futurs travaux

Pour ce qui est de PARLOG, nous n'avons pas évoqué le problème des gardes qui bouclent. Ce cas semble en relation étroite avec la propriété d'équité de l'interpréteur. Enfin, la spécification des langages voisins (GHC, concurrent Prolog) semble pouvoir être traitée par la même méthode et ne devrait pas être très éloignée de celle actuelle pour PARLOG : mis à part les problèmes de syntaxe abstraite, les principales différences tiennent aux mécanismes de définition des clauses candidates et de suspension de l'unification.

conclusion

Il s'agissait ici, comme annoncé au début du document, de réaliser une étude de faisabilité pour une spécification formelle de la sémantique de PARLOG : la seule sémantique existante pour ce langage était celle informelle de ses concepteurs [CG 86]. Excepté pour certaines figures prédéfinies du langage qui nécessiteraient un léger complément, nous pensons avoir accompli ce travail.

La programmation logique en clauses de Horn plus négation, dotée d'une sémantique purement déclarative [DF 87a], couplée avec des méthodes de preuves de correction et complétude semble un outil puissant de conception de spécification de la sémantique des langages logiques aussi bien séquentiels [DR 87]que parallèles.

Remerciements : nous tenons à remercier ici C. et P. Codognet, P. Deransart, G. Ferrand et M. Jourdan pour les discussions que nous avons eues ensemble.

Bibliographie

[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 in 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.

[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.

[GJR 87] GARCIA J., JOURDAN M., RIZK A. : An implementation of PARLOG using high-level tools, ESPRIT '87 Achievements and Impact, North-Holland.1987.

[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.

[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.

LA SEMANTIQUE DE PARLOG

DEFINITION DES STRUCTURES DE DONNEES

L'arbre et/ou sera représenté par une forêt .

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 **).**

SYNTAXE ABSTRAITE

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).

PARTIE PRINCIPALE

parlog-semantics#3 : **parlog-semantics**(P, G, F) alors **si** P est un programme et G un but **alors** F est un arbre et/ou partiel obtenu par un parcours construction de l'arbre et/ou réduit à 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) alors **si** P est un programme, F un arbre partiel et/ou **alors** F1 est un arbre partiel et/ou obtenu par un parcours construction à partir de l'arbre 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) alors **si** F est un arbre et/ou **alors** c'est un arbre succès.

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

is-a-suspended-tree#1 : **is-a-suspended-tree**(F) alors **si** F est un arbre et/ou **alors** c'est un arbre suspendu.

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

is-a-failure-tree#1 : **is-a-failure-tree**(F) alors **si** F est un arbre et/ou **alors** c'est un arbre d'échec.

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

success-leaves-only#1 : **success-leaves-only**(F) alors **si** F est un arbre et/ou **alors** toutes ses feuilles sont des feuilles succès.

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

all-success-nodes#1 : **all-success-nodes**(L) alors **si** L est une liste de nœuds **alors** tous les nœuds sont des nœuds succès.

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) alors **si** N est un nœud **alors** c'est un nœud succès.

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

suspended#2 : **suspended**(F, V) alors **si** F est un arbre et/ou partiel **alors** toutes les feuilles non succès de F sont suspendues sur la liste de variables V et il existe au moins une feuille suspendue ou bien sont des feuilles en attente.

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) alors **si** L est une liste de nœuds **alors** elle ne comporte que des nœuds suspendus, en attente ou succès.

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) alors **si** L est une liste de nœuds **alors** S est la liste des nœuds de L qui sont suspendus et ces nœuds sont suspendus sur la liste de 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) alors **si** N est un nœud **alors** il est suspendu.

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

suspended-node-on-var#2: **suspended-node-on-var**(N, V) alors **si** N est un nœud **alors** N est suspendu sur la liste de variables V.

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

is-a-failure-node#1: **is-a-failure-node**(N) alors **si** N est un nœud **alors** N est un nœud d'échec.

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

is-a-waiting-node#1: **is-a-waiting-node**(N) alors **si** N est un nœud **alors** N est en attente.

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

has-a-failure-node#1: **has-a-failure-node**(F) alors **si** F est un arbre **alors** il possède un nœud d'échec.

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

active-leaves#2: **active-leaves**(F, L) alors **si** F est un arbre **alors** L est la liste des nœuds actifs de 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) alors **si** L est une liste de nœuds **alors** L1 est la liste des nœuds actifs dans 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) alors **si** N est un nœud **alors** c'est un nœud actif.

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) alors **si** P est un programme, F un arbre et/ou partiel, N la feuille de F active **alors** F1 est l'arbre et/ou obtenu après traitement de 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) alors **si** F est un arbre et/ou et N une feuille **alors** F1 est l'arbre et/ou après la mise à jour éventuelle de la feuille en attente.

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) alors **si** L est une liste de nœuds et N un nœud de L **alors** L1 est la liste des nœuds de L situés après N qui sont en attente.

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) alors **si** L est une liste de nœuds et N un nœud de L **alors** N1 est le premier nœud en attente situé après N dans 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) alors **si** P est un programme, F est un arbre et/ou partiel et A un atome , Q un paquet de clauses définissant l'atome A et M la déclaration de modes associée **alors** L est la liste des clauses de Q qui sont candidates à la résolution de A dans le contexte de F et qui peuvent éventuellement être choisies pour la résolution de A en tenant compte de l'opérateur de recherche séquentielle.

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) alors **si** P est un programme, F est un arbre et/ou partiel et A un atome, Q un paquet de clauses définissant l'atome A et M la déclaration de modes associée **alors** V est la liste des variables de suspension pour la résolution de 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) alors **si** P est un programme, F un arbre et/ou partiel, A un atome et M la déclaration de mode associée **alors** C est une clause candidate à la résolution de A dans le contexte de 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) alors **si** P est un programme, F un arbre et/ou partiel, A un atome et M le mode associé **alors** C n'est pas une clause candidate à la résolution de A dans le contexte de 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) alors **si** B est un but et S une substitution **alors** B1 est le but B instancié par 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) alors **si** X et Y sont des termes et M une déclaration de modes associée **alors** leurs arguments d'entrée sont unifiables.

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) alors **si** P est un programme, F un arbre et/ou partiel, A un atome et M le mode associé **alors** C est un clause suspendue sur l'ensemble de variables V pour la résolution de A dans le contexte de 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) alors **si** P est un programme, F un arbre et/ou partiel, A un atome et M le mode associé **alors** C n'est pas une clause suspendue pour la résolution de A dans le programme P et dans le contexte de 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) alors **si** F est une forêt et S une substitution **alors** F1 est la forêt obtenue après instantiation par S des noeuds de 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) alors **si** N est un nœud et S une substitution **alors** M est le nœud obtenu après instantiation de 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) alors **si** P est un programme,F un arbre et/ou partiel, N une feuille active de F, C une clause candidate pour le but porté par N **alors** F1 est la forêt réduite à la forêt associée à la garde et dont les autres racines sont les fils de N, S étant le MGU du but et de la tête de la 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) alors **si** P est un programme, F un arbre et/ou, N une feuille active et C une clause candidate à la résolution du but porté par N **alors** L est la liste des noeuds portant les atomes du but B tous portant le même numéro 0.
not possible-tree(P, F, N, C) **alors** la tête de la clause candidate choisie C ne s'unifie pas avec l'atome porté par N.

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

build-list-of-node#2: **build-list-of-node**(B, L) alors **si** B est un but **alors** L est la liste des noeuds portant les atomes du but B tous portant le même numéro 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) alors **si** I est un nombre de Dewey et L une liste de noeuds **alors** F est la forêt dont les racines sont les noeuds de L numérotés à partir de 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) alors **si** F est un arbre et/ou partiel, N un nœud de F, V une liste de variables **alors** F1 est l'arbre obtenu à partir de F en suspendant N sur l'ensemble de 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) alors **si** P est un programme, G un but **alors** F est sous-arbre et/ou complet associé.

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) alors **si** P est un programme, F un arbre et/ou partiel **alors** F1 est l'arbre et/ou complet obtenu à partir de 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).

UTILITAIRES.

leaves#2 : **leaves** (F, L) alors **si** F est un arbre et/ou **alors** L est la liste des feuilles de 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) alors **si** F est un arbre et/ou **alors** N est la feuille de F la plus à droite.

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

goal#2 : **goal**(N , A) alors **si** N est un nœud **alors** A est l'atome porté par N.

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

nodenumber#2 : **nodenumber**(N , I) alors **si** N est un nœud **alors** I est le numéro de N.

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

append#3 : **append**(L1, L2, L3) alors **si** L1 et L2 sont des listes **alors** L3 est la liste concaténée de L1 et L2.

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

member#2 : **member**(X, L) alors **si** L est une liste **alors** X est un élément de L.

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

equal#2 : **equal**(X , Y) **si et seulement si** X est égal à Y.

equal (X, X).

addforest#4 : **addforest**(F, N, T, F1) alors **si** F est une forêt , N une feuille de F, T une forêt **alors** F1 est la forêt obtenue à partir de F en enracinant la forêt T en 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) alors **si** F est une forêt , N une feuille de F qui n'est pas la première, **alors** T est l'arbre dont la racine est le frère précédent N dans 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) alors **si** F est un arbre, N une feuille de F, M un nœud **alors** F1 est l'arbre obtenu à partir de F en ajoutant M comme fils de N.

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

ready#3 : **ready**(F, N, F1) alors **si** F est une forêt et N un nœud en attente de F, **alors** F1 est la forêt égale à F excepté que le correspondant de N n'est pas en attente.

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) alors **si** F1 et F2 sont des forêts **alors** F3 est la concaténation de F1 et 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).

DENOTATION RELATIVE

L-unify#5 : **L-unify**(A, B, M, S, V) pour tous les littéraux A et B, toutes les déclarations de modes M, toutes les substitutions S tels que S est un MGU de la liste des termes input de A et B, et V est l'ensemble des variables suspendues.(éventuellement V est vide)

L-unify#3 : **L-unify**(A, B, S) pour tous les littéraux A et B,toutes les substitutions S tels que S est un MGU de A et B.

L-first-active-leaf#3: **L-first-active-leaf**(L, N) pour toutes les listes de nœuds actifs et les nœuds N tels que N est le 1er nœud qui éxécute le commit operator ou bien se suspend.

L-instance#3: **L-instance**(T, S, T1) pour tous termes T et T1, toute substitution S tels que T1 est l'instance de T par S.

L-literal#1 : **L-literal**(A) pour toutes les représentations de littéraux du dialecte PARLOG.

L-mode#2 : **L-mode**(Q, M) pour toutes les représentations de paquets Q de PARLOG et toutes les représentations de modes possibles associés.

L-packet#4 : **L-packet**(P, A, M, Q) pour tous les littéraux A, tous les programmes P et Q , toutes les déclarations de modes M tels que Q est le paquet de clauses correspondant à A dans le programme P et M la déclaration de modes associée.

L-random-choice#2 : **L-random-choice**(S, C) pour tous les ensembles de clauses S et les clauses C telles que C est un choix au hasard dans S .

L-rename#3 : **L-rename**(F, T, T1) pour toute forêt F, tous termes T et T1 tel que T1 est un renommage de T avec des variables n'apparaissant pas dans F.

L-same-name#2 : **L-same-name**(F, S) pour toutes les clauses C et tous les paquets P tels que la clause C définiit le même prédicat que P.

L-subs#2 : **L-subs**(F, S) pour tous les arbres F et/ou succès et les représentations de substitutions S tels que S soit la substitution réponse induite par F.

L-substitution#1 : **L-substitution**(S) pour toutes les représentations de substitutions S.

L-update#3 : **L-update**(L, S, L1) pour toutes les listes d'ensembles de variables L et L1, toutes les représentations de substitutions S telles que L1 soit constituée des ensembles résultant de la mise à jour relativement à S des éléments de L si aucun de ces résultats n'est vide sinon L1 est vide.
Un ensemble de variables V1 est le résultat de la mise à jour d'un ensemble de variables V relativement à une substitution S si V1 est constitué des éléments de V qui ne sont pas dans le support de S (V1 est donc vide si S affecte toutes les variables de V).

L-variable#1 : **L-variable**(V) pour toutes les variables V du dialecte PARLOG.

Remarques :

Le prédicat **L-packet** se définit simplement dès que l'on choisit une représentation des littéraux, ce qui n'est pas nécessaire.

Le prédicat **L-random-choice** qui exprime que son 2ième argument est un élément choisi au hasard dans son 1er argument est en fait dépendant de "l'implémentation du hasard" qui est faite dans l'interpréteur : dans le cas supposé de l'implémentation d'un interpréteur équitable, ce prédicat spécifie simplement l'appartenance d'un élément à un ensemble.

Le prédicat **L-first-active-leaf** est lui aussi dépendant de l'implémentation : dans le cas supposé de l'implémentation d'un interpréteur équitable, ce prédicat spécifie simplement l'appartenance d'un élément à un ensemble.

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