Présentation simplifiée d'une machine de gestion de mémoire pour les interpréteurs Prolog

Yves Bekkers, Bernard Canet, Olivier Ridoux, Lucien Ungaro

▶ **To cite this version:**

Yves Bekkers, Bernard Canet, Olivier Ridoux, Lucien Ungaro. Présentation simplifiée d'une machine de gestion de mémoire pour les interpréteurs Prolog. [Rapport de recherche] RR-0280, INRIA. 1984. inria-00076278

HAL Id: inria-00076278
https://inria.hal.science/inria-00076278

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.: (3) 954 90 20

Rapports de Recherche

N° 280

PRÉSENTATION SIMPLIFIÉE D'UNE MACHINE DE GESTION DE MÉMOIRE POUR LES INTERPRÉTEURS PROLOG

Yves BEKKERS
Bernard CANET
Ollivier RIDOUX
Lucien UNGARO

Mars 1984

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

PRESENTATION SIMPLIFIEE D'UNE

MACHINE DE GESTION DE MEMOIRE POUR LES INTERPRETEURS PROLOG

Publication Interne n° 224

Février 1984

20 pages

Yves BEKKERS, Bernard CANET, Ollivier RIDOUX, Lucien UNGARO

I.R.I.S.A. - Campus Universitaire de Beaulieu

Avenue du Général Leclerc

35042 RENNES CEDEX FRANCE

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

PRESENTATION SIMPLIFIEE D'UNE

MACHINE DE GESTION DE MEMOIRE POUR LES INTERPRETEURS PROLOG

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
I.R.I.S.A., Campus Universitaire de Beaulieu
Avenue du Général Leclerc - 35042 RENNES Cedex - FRANCE

Résumé

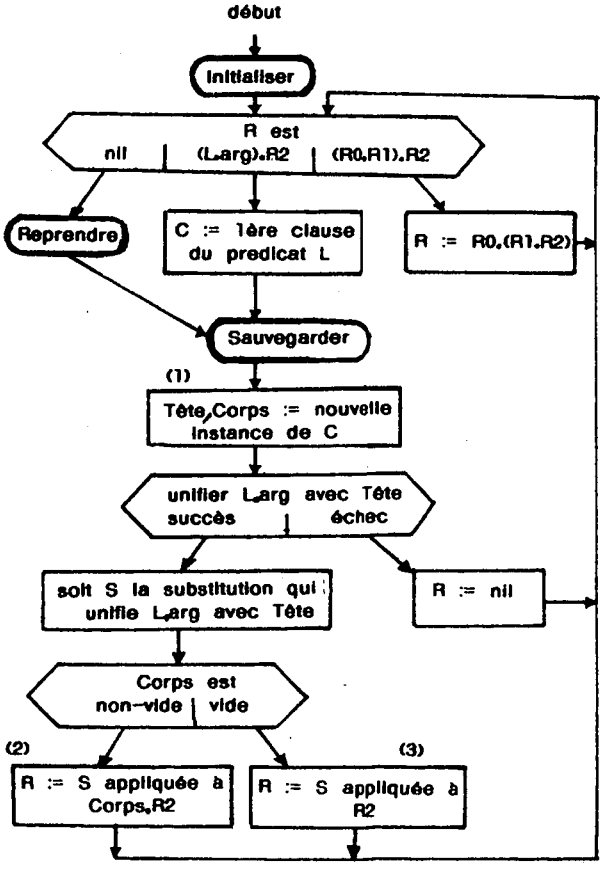
Cet article décrit une machine qui assure la gestion de l'état d'un interpréteur PROLOG. Cette machine comporte un récupérateur de mémoire basé sur une méthode originale d'interprétation de l'état de liaison des variables. Le récupérateur tient compte de la dimension spécifique due à l'indéterminisme de PROLOG et réalise ainsi une détection des cellules inutiles plus complète que ne le font les systèmes existant, hérités de ceux mis en oeuvre pour LISP.

PAPIER RÉCUPÉRÉ ET RECYCLÉ

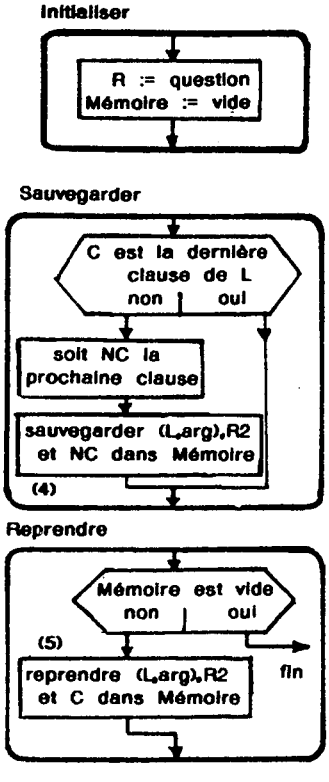
1. Un interpréteur PROLOG

L'interpréteur PROLOG proposé, détaillé en figure 1, est la transcription directe du principe de résolution. Les résolvantes y sont manipulées objectivement en tant que termes binaires.
Une résolvante est soit *nil*, auquel cas elle est la résolvante vide, soit un terme construit *(R1,R2)*, où *R1* est une résolvante non vide et *R2* est une résolvante. Une résolvante non vide est soit un litteral *(L,arg)*, où *L* est un atome qui identifie un prédicat et *arg* est un terme binaire qui tient lieu d'argument du littéral, soit un terme construit *(R11,R12)*, où *R11* et *R12* sont des résolvantes non vides.

La figure 1.1 montre la transformation répétitive de la résolvante courante. On y voit les besoins de "création de variables" en (1) et les besoins de "construction de termes", "sélection de sous-termes" et "substitution de variables" en (2) et (3). La figure 1.2 montre la gestion de l'indéterminisme par retour arrière. On y voit les besoins de "sauvegarder" des résolvantes en (4) et de les "reprendre" en (5). Ces opérations de base sont prises en charge par une machine intermédiaire utilisée par l'interpréteur PROLOG, désormais appelé "l'utilisateur".



1.1 Réécriture de la résolvante courante.



1.2 Gestion de l'indéterminisme.

figure 1 – L'interpréteur PROLOG.

2. La machine intermédiaire

La machine intermédiaire offre un état formé d'une "composante vive", qui permet à l'utilisateur de désigner par des "noms" les sous-termes de la résolvante courante, et d'une "sauvegarde", qui est une liste ordonnée de termes, chacun d'entre eux étant une résolvante sauvegardée. La machine assure la correspondance entre les noms et les termes.

2.1. Les commandes

L'utilisateur émet des commandes, à l'occasion desquelles il échange des noms avec la machine. L'utilisateur ne connait initialement aucun nom, hormis ceux des atomes. Il obtient d'autres noms au moyen des commandes. Les commandes qui ont des effets sur la gestion de mémoire sont

construire(*ng,nd*:**nom**) :**nom**
créer_variable :**nom**
substituer(*nv,nt*:**nom**)
sauvegarder(*n*:**nom**)
reprendre :**nom**
réduire(*n*:**nom**).

Le résultat de **construire** est un nom pour le terme construit dont les sous-termes de droite et de gauche sont respectivement désignés par les noms *ng* et *nd*. Comme pour LISP, la machine dispose de commandes, non décrites ici, pour consulter les composants des termes binaires.

Les commandes restantes concernent les concepts de variable et d'indéterminisme propres à PROLOG. Le résultat de **créer_variable** est le nom d'une nouvelle variable. La commande **substituer** fait correspondre des termes plus instanciés aux noms dans la composante vive: après cette commande, chaque nom désigne le terme désigné auparavant, dans lequel la variable désignée par *nv* a été remplacée par le terme désigné par *nt*. La commande **sauvegarder** empile le terme désigné par *n* dans la sauvegarde, la composante vive demeurant inchangée. Le résultat de **reprendre** est un nom pour le terme dépilé de la sauvegarde. Cette désignation devient la nouvelle composante vive.

Après une commande **réduire**, la composante vive est réduite à la désignation de *n*. L'invocation de cette commande permet à la machine de récupérer les cellules de mémoire qui ne participent ni à la représentation du terme désigné par *n* ni à la représentation d'un terme sauvegardé. Ces cellules sont rendues disponibles pour une utilisation ultérieure. La révèlation explicite à la machine des accès dont dispose l'utilisateur est une technique plus souple que l'utilisation d'un nombre fixe de registres d'accès connus par la machine: l'utilisateur peut conserver temporairement dans ses mémoires privées un nombre illimité de noms qui sont des accès dans la composante vive, par exemple pendant l'unification.

2.2. Mise en oeuvre

La mise en oeuvre de la machine supporte deux processus: le "processus utilisateur" qui soumet les commandes, et un "processus récupérateur", interne, qui fonctionne en parallèle pour récupérer les ressources de mémoire devenues inutiles à la représentation.

3. La représentation

Les types d'information basiques sont **référence**, **donnée** et **nom**. La machine possède une mémoire qui est une collection de cellules, chacune adréssée par une **référence**. Toute cellule peut être considérée sous l'un quelconque des formats **constructeur**, **variable** ou **géniteur**.
Dans ce qui suit, ↑*ref* dénote l'accès à la cellule de référence *ref*, et **a**:*data*, **c**:*ref* et **v**:*ref* dénotent les diverses sortes de noms.

référence = {**réf_nulle**} ∪ {1..**maxref**} ; **donnée** = {0..**maxdonnée**}

nom = structure
| **indicateur** :{**a**,**c**,**v**}
| **information** : **référence** ou **donnée**

constructeur = structure
| **gauche** :**nom**
| **droite** :**nom**

variable = structure
| **statut** :{**libre**,**indécis**}
| **niveau** :**référence**
| **liaison** :**nom**

géniteur = structure
| **nature** :{**actif**,**mort**}
| **inférieur** :**référence**
| **nom** :**nom**

3.1 Les termes

Les termes sont représentés comme suit:
- Le nom **a**:*data* est le nom direct de l'atome *data*.
- Le nom **c**:*ref*, où *ref* est la référence d'une cellule constructeur contenant un nom du terme *tg* dans son champ gauche et un nom du terme *td* dans son champ droite, est un nom direct du terme (*tg*.*td*).
- Le nom **v**:*ref* est soit le nom direct d'une variable, si *ref* est la référence d'une cellule variable libre, soit un nom indirect du terme *tl*, si *ref* est la référence d'une cellule variable liée qui contient un nom de *tl* dans son champ liaison. L'état libre ou lié d'une cellule variable est déterminé différemment selon qu'elle est considérée au titre de la composante vive ou au titre d'un terme sauvegardé.

3.2. Les liaisons dans la composante vive

Considérée au titre de la composante vive, une cellule variable est libre si son champ statut contient **libre** ou si le champ nature du géniteur référencé par son champ niveau contient **mort**. Lorsqu'une cellule est allouée pour une nouvelle variable, son champ statut est initialisé à **libre**. A l'issue d'une substitution portant sur une variable, le champ statut de la variable contient **indécis** et son champ niveau contient la référence d'un géniteur actif. Ceci rend liée la variable. Elle demeure liée jusqu'à ce que l'exécution d'une commande **reprendre** altère l'état de son géniteur en inscrivant **mort** dans son champ nature, ce qui rend à nouveau libre la variable. Les géniteurs jouent le rôle habituel de la traînée dans les interpréteurs PROLOG.

3.3. Les liaisons dans les termes sauvegardés

Les géniteurs actifs sont ordonnés selon une liste définie par leur champ inférieur. Chaque géniteur définit un "niveau". Le géniteur le plus bas définit le niveau 0 et contient **réf_nulle** dans son champ inférieur. Un terme sauvegardé est toujours associé à un niveau et il est désigné par le champ nom du géniteur correspondant. Ce terme résulte d'une interprétation de l'état de liaison des

variables qui tient compte du niveau: une cellule variable est libre dans la représentation d'un terme sauvegardé de niveau *k* si son champ statut contient **libre** ou si le géniteur référencé par son champ niveau contient **mort** dans son champ nature ou est de niveau supérieur à *k*.
Cette interprétation des liaisons est utilisée par le récupérateur de mémoire pour reconstituer les accès exacts aux cellules et déterminer leur utilité réelle. La plupart des systèmes existant, dérivés de ceux mis en oeuvre pour LISP, ne prennent pas en compte la nouvelle dimension introduite par l'indéterminisme et interprètent les liaisons de variables indépendamment du niveau de la résolvante observée. Ceci revient à considérer une résolvante plus instanciée qu'elle ne l'est, et conduit à conserver plus de cellules que nécessaire.

4. Le processus utilisateur

Le registre **niveau_utilisateur** contient la référence du géniteur de plus grand niveau. A chaque commande correspond une procédure. La procédure **recherche_représentant** traverse les liaisons de variables pour délivrer le nom direct équivalent à un nom donné. Ceci introduit des chaines de liaisons plus courtes et accroit les occasions de perte d'accès aux variables.
La commande **substituer** signe la liaison d'une variable par la référence du géniteur de plus grand niveau. Ceci a pour effet de valider cette liaison dans la composante vive, tant que le géniteur demeure actif. En conséquence, la composante vive est en permanence associée au plus grand niveau. La commande **sauvegarder** sauvegarde un terme donné en rangeant son nom dans le géniteur de plus grand niveau et crée un géniteur de niveau supérieur qui servira de signature pour les liaisons ultérieures dans la composante vive. La commande **reprendre** provoque la mort du géniteur de plus grand niveau. Ceci détruit les liaisons de variables selon leur interprétation dans la composante vive. La commande restitue alors l'association entre la composante vive et le géniteur immédiatement inférieur, et délivre le nom du terme sauvegardé associé. La commande **réduire** déclenche le récupérateur de mémoire, en lui soumettant un nom qui constitue le seul accès conservé par l'utilisateur. La tête de la liste des géniteurs est automatiquement transmise au récupérateur, et définit la racine des accès dûs aux termes sauvegardés.

```
niveau_utilisateur :référence

procédure créer_variable :nom
| ref_v:=allocation_cellule
| ↑ref_v.statut:=libre
| résultat v.ref_v

procédure construire(n_gauche,n_droite:nom) :nom
| ref_cons:=allocation_cellule
| ↑ref_cons.gauche:=recherche_représentant(n_gauche)
| ↑ref_cons.droite:=recherche_représentant(n_droite)
| résultat c.ref_cons

procédure substituer(nv,nt:nom)
| ref_v:=recherche_représentant(nv).information
| ↑ref_v.liaison:=recherche_représentant(nt)
| lier_variable(ref_v)

procédure sauvegarder(n:nom)
| ↑niveau_utilisateur.nom:=recherche_représentant(n)
| ref_g:=allocation_cellule
| ↑ref_g.inférieur:=niveau_utilisateur
| ↑ref_g.nature:=actif
| niveau_utilisateur:=ref_g

procédure reprendre :nom
| recalage
| libérer_variables
| niveau_utilisateur:=↑niveau_utilisateur.inférieur
| résultat ↑niveau_utilisateur.nom

procédure réduire(n:nom)
| démarrage_fournée(n)

procédure recherche_représentant(n:nom) :nom
| nn:=n
| boucle tantque nn.indicateur=v et test_liaison_absolue(nn.information)=liée
| | nn:=↑(nn.information).liaison
| résultat nn
```

5. Le processus récupérateur de mémoire

Le récupérateur de mémoire procède cycliquement. Chaque cycle est appelé une "fournée" et consiste en une phase de marquage suivie d'une phase de ramassage.
Le registre **niveau_marquage** contient la référence du géniteur qui définit le niveau couramment sous marquage. Le registre **nom_marquage** contient le nom qui constitue la racine des accès aux cellules représentant le terme sauvegardé à parcourir.

```
nom_marquage :nom ;  niveau_marquage :référence

processus récupérateur
| bloc marquage
| | boucle tantque niveau_marquage≠réf_nulle
| | | marquage_terme(nom_marquage); descente_niveau
| bloc ramassage
| | ref_cel:=0
| | boucle tantque ref_cel<maxref
| | | ref_cel:=ref_cel+1
| | | relâche_cellule(ref_cel)
| attente_fournée

procédure marquage_terme(n:nom)
| ref:=n.information
| si n.indicateur≠a et test_marque(ref)=non_marquée alors
| | cas n.indicateur
| | c alors marquage_terme(↑ref.gauche); marquage_terme(↑ref.droite)
| | v alors si test_liaison_relative(ref)=liée alors marquage_terme(↑ref.liaison)
| | marquage_cellule(ref)

procédure lier_variable(ref_v:référence) exclusion liaisons
| ↑ref_v.niveau:=niveau_utilisateur; ↑ref_v.statut:=indécis

procédure libérer_variables exclusion liaisons
| ↑niveau_utilisateur.nature:=mort

procédure test_liaison_absolue(ref_v:référence) :{libre,liée} exclusion liaisons
| si ↑ref_v.statut=libre ou ↑(↑ref_v.niveau).nature=mort alors résultat libre
| sinon résultat liée

procédure test_liaison_relative(ref_v:référence) :{libre,liée} exclusion liaisons
| si ↑ref_v.statut=libre alors résultat libre
| sinon
| | ref_g:=↑ref_v.niveau
| | cas test_marque(ref_g)
| | marquée alors résultat libre
| | non_marquée alors
| | | si ↑ref_g.nature=mort alors ↑ref_v.statut:=libre; résultat libre
| | | sinon résultat liée

procédure descente_niveau exclusion position
| décrémentation_niveau

procédure recalage exclusion position
| si niveau_marquage=niveau_utilisateur alors
| | suspension récupérateur
| | décrémentation_niveau
| | recommencement récupérateur

procédure décrémentation_niveau
| marquage_cellule(niveau_marquage)
| niveau_marquage:=↑niveau_marquage.inférieur
| si niveau_marquage≠réf_nulle alors
| | nom_marquage:=↑niveau_marquage.nom
```

5.1. Marquage des niveaux par ordre décroissant

La phase de marquage procède niveau après niveau par ordre décroissant. De ceci découle la propriété importante que le marquage visite exactement une fois chaque cellule utile, lors du marquage du niveau le plus grand qui accède à cette cellule. Ceci peut être informellement justifié comme suit. Soient *i* et *j* deux niveaux, avec *i*<*j*. Toute cellule **constructeur** donne les mêmes accès immédiats sous ces deux niveaux. Toute cellule **variable** donne sous le niveau *i* un accès immédiat soit vide soit identique à l'accès qu'elle donne sous le niveau *j*, car une variable liée sous un niveau donné demeure identiquement liée sous tous les niveaux supérieurs. Par conséquent, l'ensemble des cellules accessibles au niveau *i* depuis une cellule donnée est inclus dans l'ensemble des cellules accessibles au niveau *j* depuis cette cellule.

Cette propriété est illustrée sur la figure 2. Les ensembles de cellules E1 et E0 représentent respectivement les termes de niveaux 1 et 0, conservés en association avec les géniteurs C11 et C13. Lors du marquage du niveau 0, il n'est pas nécessaire de parcourir les cellules au-delà de la cellule C5, déjà visitée au niveau 1, car toutes les cellule auxquelles ce parcours conduirait sont déjà marquées.
Ne pas prendre en compte les niveaux conduirait à considérer à tort les cellules C3 et C4 comme utiles.

Après le marquage d'un niveau, la procédure **descente_niveau** marque le géniteur correspondant. Ainsi, la comparaison entre le niveau des liaisons et le niveau sous marquage revient simplement à tester la marque d'allocation des géniteurs.

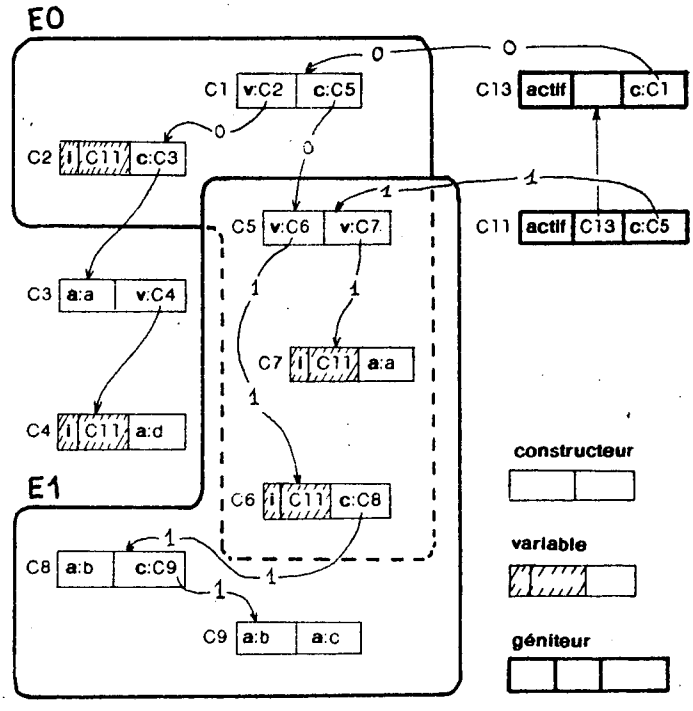


figure 2 - Parcours des cellules

5.2 Incidence du parallélisme sur le marquage

L'exécution de **reprendre** par le processus utilisateur peut provoquer la destruction logique du niveau couramment sous marquage. Dans ce cas, le marquage est abandonné et le récupérateur entame aussitôt le marquage du niveau inférieur. L'abandon d'un marquage est possible si les cellules déjà marquées par le récupérateur ne donnent pas accès à des cellules non marquées. Pour assurer cette condition, le marquage des cellules est fait en revenant des feuilles vers la racine.

L'exécution parallèle du processus utilisateur n'affecte pas le principe consistant à considérer comme feuilles les cellules marquées. En effet, les cellules allouées depuis le début de la fournée courante conduisent à des cellules qui ont été allouées soit au cours de cette fournée, auquel cas elles ont été marquées au moment de l'allocation, soit avant cette fournée, auquel cas elles sont accessibles à partir de la racine confiée au récupérateur de mémoire en début de fournée, et seront marquées lors du parcours issu de cette racine.

6. Allocation des cellules, fournées

Le tableau **statut_alloc** conserve le statut d'allocation de chaque cellule. Il y a trois statuts possibles: **disponible**, qui qualifie une cellule disponible, et les deux indices de fournée 0 et 1, qui qualifient les cellules allouées. Le registre **fournée_courante** contient l'indice de la fournée courante. Cet indice est utilisé comme marque pour les cellules au moment de leur allocation ou lorsqu'elles sont rencontrées par le récupérateur pendant sa phase de marquage.

```
fournée_courante :{0,1}
récupérateur_en_attente :{vrai,faux}

procédure attente_fournée exclusion fournées
| récupérateur_en_attente:=vrai; suspension récupérateur

procédure démarrage_fournée(n:nom) exclusion fournées
| si récupérateur_en_attente alors
| | fournée_courante:=(fournée_courante+1) mod 2
| | niveau_marquage:=niveau_utilisateur; nom_marquage:=n
| | récupérateur_en_attente:=faux; recommencement récupérateur


procédure marquage_cellule(ref:référence)
| statut_alloc[ref]:=fournée_courante

procédure test_marque(ref:référence) :{marquée,non_marquée}
| si statut_alloc[ref]=fournée_courante alors résultat marquée
| sinon résultat non_marquée


cellules_disponibles :référence
statut_alloc :tableau 1..maxref de {0,1,disponible}

procédure allocation_cellule :référence exclusion allocation
| si cellules_disponibles=réf_nulle alors attente cellule_disponible
| marquage_cellule(cellules_disponibles); résultat cellules_disponibles
| cellules_disponibles:=↑cellules_disponibles.suivante

procédure relâche_cellule(ref_cel:référence) exclusion allocation
| si statut_alloc[ref_cel]=(fournée_courante+1)mod 2 alors
| | statut_alloc[ref_cel]:=disponible
| | ↑ref_cel.suivante:=cellules_disponibles; cellules_disponibles:=ref_cel
| | signal cellule_disponible
```

Les fournées

Les fournées sont introduites dans la gestion de mémoire pour permettre le parallélisme entre l'utilisateur et le récupérateur de mémoire. Au début de la i-ème fournée, le statut d'allocation de toute cellule non disponible est i mod 2. La i-ème fournée est associée à l'indice $(i+1)$ mod 2. Pendant une fournée, son indice est inscrit dans le statut d'allocation des cellules devenant allouées ou rencontrées durant la phase de marquage du récupérateur. Après la phase de marquage, les cellules qui ont conservé le statut d'allocation i mod 2 de la fournée précédente peuvent être rendues disponibles. Ceci est réalisé par la phase de ramassage, après quoi une nouvelle fournée peut commencer.

7. Extensions

La machine présentée a été simplifiée de façon à faire apparaitre les aspects originaux d'un récupérateur de mémoire basé sur la stratification apportée par l'indéterminisme dans l'utilisation de la mémoire. Dans la machine réelle, les algorithmes sont étendus pour prendre en charge la primitive "couper" de PROLOG, dont l'effet logique est de détruire des points de reprises sauvegardés, et qui est une cause majeure de perte d'accès. La récupération de mémoire est alors pleinement mise en valeur.

Un simulateur de la machine intermédiaire et un interpréteur PROLOG qui l'utilise sont en cours de développement. Une réalisation matérielle composée de deux processeurs est à l'étude. L'un des processeurs sera microprogrammé pour réaliser la majeure partie de la machine intermédiaire.

Références

[1] "Implementing Prolog-compiling logic programs", D.H.D. Warren, D.A.I. Research Report, No. 39 and 40, University of Edinburgh, 1977.

[2] "List Processing in Real Time on a Serial Computer", H.G. Baker, Communications of the ACM, Vol. 21 No. 4 280-294, April 1978.

[3] "On-the-Fly Garbage Collection: An Exercise in Cooperation", E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, Communications of the ACM, Vol. 21 No. 11 966-975, Nov. 1978.

[4] "The memory management of PROLOG implementations", M. Bruynooghe, in logic programming eds Tarnlund and Clark, Academic press 1981.

[5] "Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques" Version 1, Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro, publication interne IRISA No 222, Janvier 1984.

A SIMPLE MEMORY MANAGEMENT MACHINE

FOR PROLOG INTERPRETERS

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
I.R.I.S.A., Campus Universitaire de Beaulieu
Avenue du Général Leclerc - 35042 RENNES Cedex - FRANCE

Abstract

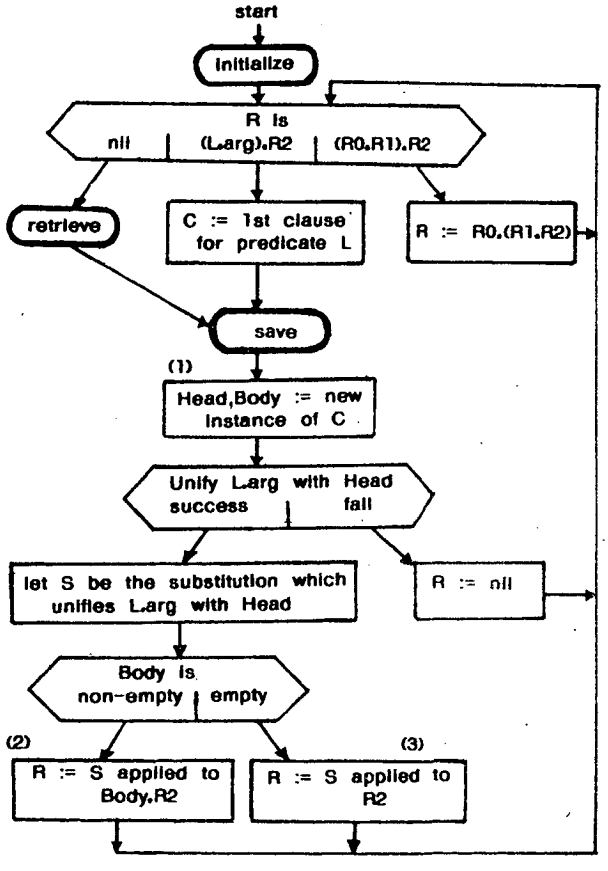
This paper describes a machine for the management of a PROLOG interpreter state. The machine includes a garbage-collector which uses an original algorithm to decide of the variables bindings. It takes into account the specific PROLOG dimension due to indeterminism and thus leads to a more complete detection of useless cells than present-day systems, inherited from LISP garbage--collectors.

1. A PROLOG interpreter

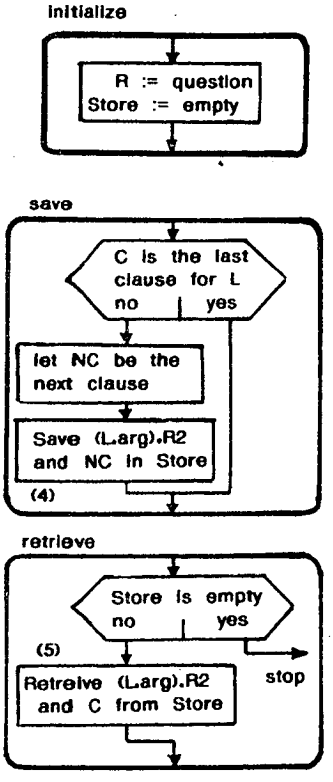
The PROLOG interpreter, detailed in figure 1, is a straighforward transcription of the resolution principle. Goal statements are objectively manipulated as binary terms.

A goal statement is either *nil*, in which case it is the empty goal statement, or a construct *(R1.R2)*, where *R1* is a non-empty goal statement and *R2* is a goal statement. A non-empty goal statement is either a goal *(L.arg)*, where *L* is an atom identifying a predicate and *arg* is a binary term standing as the argument of the goal, or a construct *(R11.R12)*, where *R11* and *R12* are non-empty goal statements.

Figure 1.1 shows the cyclic rewriting of the current goal statement. The need to "create variables" appears in (1). "terms construction", "sub-terms selection" and "variables substitution" are needed in (2) and (3). Figure 1.2 shows the management of indeterminism using backtracking. The need to "save" goal statements appears in (4) and the need to "retrieve" them appears in (5). These basic operations have been implemented on an intermediate machine used by the PROLOG interpreter, hencefoward called "the user".



1.1 rewriting the current goal statement



1.2 management of indeterminism

figure 1 - The PROLOG interpreter.

2. The intermediate machine

The intermediate machine has a state composed of a "top-level", which allows the user to signify sub-terms of the current goal statement by means of "names", and a "store", which is an ordered set of terms, each of them being a saved goal statement. The machine keeps the correspondance between names and terms.

2.1. Commands

The user invokes commands, by means of which it exchanges names with the machine. The user initially knows no names but those of atoms. It gets other names by means of commands. The commands connected with memory management are

construct(*ln*,*rn*:**name**) :**name**
create_variable :**name**
substitute(*vn*,*tn*:**name**)
save(*n*:**name**)
retrieve :**name**
reduce(*n*:**name**).

The result of **construct** is a name for a term the left and right sub-terms of which are respectively signified by the names *ln* and *rn*. Like in LISP, the machine has some commands, not described here, to obtain the components of binary terms.

The remaining commands are related to PROLOG's concepts of variable and indeterminism. The result of **create_variable** is a name for a new variable. The **substitute** command makes names in the top-level signify more instanciated terms: after the command, every name signifies the term signified before, where the variable signified by *vn* is replaced by the term signified by *tn*. The **save** command pushes the term signified by *n* in the store, the top-level remaining unaffected. The result of **retrieve** is a name for the term popped from the store. This significance becomes the top-level.

After the **reduce** command, the top-level is reduced to the significance of *n*. Invoking this command allows the machine to collect every cell which does not participate to the representation of either the term signified by *n* or the saved terms. The collected cells are rendered available for future use. Instructing the machine about the user's accesses is a more flexible technique than using a fixed number of access registers known by the machine: the user can store temporarily in its own memory an unlimited number of names which are accesses in the top-level, for example during unification.

2.2. Implementation

The implementation of the machine supports two processes: the "user process" which invokes commands, and an internal "garbage-collector process" which works in parallel at the recovery of the memory resources no longer useful for the representation.

3. The representation

The basic data types are **reference**, **data** and **name**. The machine has a memory which is a collection of cells, each one addressed by a **reference**. Every cell can be considered under any of the formats **construct**, **variable** or **guardian**.

```
reference = {null_ref}∪{1..maxref} ;    data = {0..maxdata}

name = structure
  | logical : {a,c,v}
  | information : reference ... data

construct = structure
  | left :name
  | right :name

variable = structure
  | status :{free,uncertain}
  | level :reference
  | binding :name

guardian = structure
  | nature :{live,dead}
  | lower_level :reference
  | name :name
```

In the following, ↑*ref* denotes the access to the cell referenced by *ref*, and **a**:*data*, **c**:*ref* and **v**:*ref* denote the various kinds of names.

3.1 Terms

Term representation is as follows:
- The name **a**:*data* is the direct name of the atom *data*.
- The name **c**:*ref*, where *ref* is the reference of a construct cell which holds a name for *lt* in its left field and a name for *rt* in its right field, is the direct name of the term (*lt*.*rt*).
- The name **v**:*ref* is either the direct name of a variable if *ref* references a free variable cell, or an indirect name for the term *bt* if *ref* references a bound variable cell which holds a name for *bt* in its binding field. The free or bound state of a variable cell is determined differently, whether it is considered from the top-level or from a saved term.

3.2. The top-level bindings

Considered from the top-level, a variable cell is free either if its status field holds **free** or if the nature field of the guardian referenced by its level field holds **dead**. When a cell is allocated for a new variable, its status field is initialized **free**. After variable substitution, the variable's status field holds **uncertain** and its level field holds the reference of a live guardian. This makes the variable bound. The variable remains bound until a **retrieve** command alters its guardian by storing **dead** in its nature field, which makes the variable free again. The guardians play the role of the trail in PROLOG interpreters.

3.3. The saved terms bindings

The live guardians are ordered in a list defined by their lower_level field. Each guardian defines a "level". The lowest guardian defines level 0 and holds **null_ref** in its lower_level field. A saved term is always associated to a level and is signified by the name field of the corresponding guardian. This term results from an interpretation of the binding state of variables which takes the level into account: a variable cell is free in the representation of the saved term of level *k* either if its status field holds **free** or if the guardian referenced by its level field holds **dead** in its nature field or has a level greater than *k*.

This binding interpretation is used by the garbage-collector in order to find accurate accesses to cells and determine their real usefullness. Most of existing systems, derived from those implemented for LISP, do not take into account the new dimension introduced by indeterminism and interpret the bindings of variables independently of the level of the observed goal statement. This amounts to consider over-instanciated goal statements, and leads to retain more cells than necessary.

4. The user process

The **user_level** register contains the reference of the guardian at the highest level. Each command has a corresponding procedure. The **search_direct_name** procedure goes through variable bindings to yield the direct name equivalent to a given name. This induces shorter chains of bound variables and increases the opportunities of loss of access to variables.

The **substitute** command subscripts the binding of a variable with the reference of the guardian at the highest level. The effect is to validate this binding in the top-level, as long as the guardian remains alive. Therefore, the top-level is always associated with the highest level. The **save** command saves a given term by storing its name in the guardian at the highest level and creates a higher level guardian which will be used as subscript for the next bindings in the top-level. The **retrieve** command causes the death of the guardian at the highest level. This undoes the variable bindings according to the top-level interpretation. Then the command restores at the top-level the guardian right beneath, and returns the name of the associated saved term. The **reduce** command triggers the garbage-collector, supplying it with a name which constitutes the only access kept by the user. The head of the guardians list is automaticaly passed to the collector, and defines the root of the accesses due to saved terms.

```
user_level :reference

procedure create_variable :name
| ref_v:=cell_allocation
| ↑ref_v.status:=free
| result v.ref_v

procedure construct(left_n,right_n:name) :name
| ref_cons:=cell_allocation
| ↑ref_cons.left:=search_direct_name(left_n)
| ↑ref_cons.right:=search_direct_name(right_n)
| result c.ref_cons

procedure substitute(nv,nt:name)
| ref_v:=search_direct_name(nv).information
| ↑ref_v.binding:=search_direct_name(nt)
| bind_variable(ref_v)

procedure save(n:name)
| ↑user_level.name:=search_direct_name(n)
| ref_g:=cell_allocation
| ↑ref_g.lower_level:=user_level
| ↑ref_g.nature:=live
| user_level:=ref_g

procedure retrieve :name
| re_adjust_level
| unbind_variables
| user_level:=↑user_level.lower_level
| result ↑user_level.name

procedure reduce(n:name)
| start_batch(n)

procedure search_direct_name(n:name) :name
| nn:=n
| loop while nn.indicator=v and test_absolute_binding(nn.information)=bound
| ! nn:=↑(nn.information).binding
| result nn
```

5. The garbage-collector process

The garbage-collector works cyclicaly. Each cycle is called a "batch" and consists of a marking phase followed by a collecting phase.
The **marking_level** register references the guardian which defines the level currently under marking. The **marking_name** register contains the name which is the root access to the cells representing the saved term to be walked through.

```
marking_name :name ;   marking_level :reference

process garbage_collector
| block marking_phase
| | loop while marking_level≠null_ref
| | | mark_term(marking_name); down_one_level
| block collecting_phase
| | ref_cel:=0
| | loop while ref_cel<maxref
| | | ref_cel:=ref_cel+1
| | | make_cell_available(ref_cel)
| wait_next_batch


procedure mark_term(n:name)
| ref:=n.information
| if n.indicator≠a and test_mark(ref)=unmarked then
| | case n.indicator
| | c then mark_term(↑ref.left); mark_term(↑ref.right)
| | v then if test_relative_binding(ref)=bound then mark_term(↑ref.binding)
| | cell_marking(ref)
```

```
procedure bind_variable(ref_v:reference) exclusion bindings
| ↑ref_v.level:=user_level; ↑ref_v.status:=uncertain

procedure unbind_variables exclusion bindings
| ↑user_level.nature:=dead

procedure test_absolute_binding(ref_v:reference) :{free,bound} exclusion bindings
| if ↑ref_v.status=free or ↑(↑ref_v.level).nature=dead then result free
| else result bound

procedure test_relative_binding(ref_v:reference) :{free,bound} exclusion bindings
| if ↑ref_v.status=free then result free
| else
| | ref_g:=↑ref_v.level
| | case test_mark(ref_g)
| | marked then result free
| | unmarked then
| | | if ↑ref_g.nature=dead then ↑ref_v.status:=free; result free
| | | else result bound
```

```
procedure down_one_level exclusion position
| decrement_level

procedure re_adjust_level exclusion position
| if marking_level=user_level then
| | suspend garbage_collector
| | decrement_level
| | restart garbage_collector

procedure decrement_level
| cell_marking(marking_level)
| marking_level:=↑marking_level.lower_level
| if marking_level≠null_ref then
| | marking_name:=↑marking_level.name
```

5.1. Marking levels in decreasing order

The marking phase proceeds level by level in decreasing order. This yields the important property that the marking visits each usefull cell exactly once, when marking the highest level which has access to this cell. This can be informaly justified as follows. Let *i* and *j* be two levels, with *i*<j. Any **construct** cell yields the same immediate accesses for these two levels. Any **variable** cell yields at level *i* an immediate access either empty or identical to the access it yields at level *j*, because a variable bound at a given level remains identically bound for any higher level. Therefore, the set of cells accessed via a given cell at level *i* is included in the set accessed via this cell at level *j*.

This property is illustrated on figure 2. The sets of cells S1 and S0 respectively represent the terms of levels 1 and 0, kept in association with the guardians C11 and C13. While marking level 0, it is not necessary to go through the cells beyond cell C5, already visited at level 1, because all the cells this would lead to are already marked.

Not taking the levels into account would lead to wrongly consider cells C3 and C4 useful.

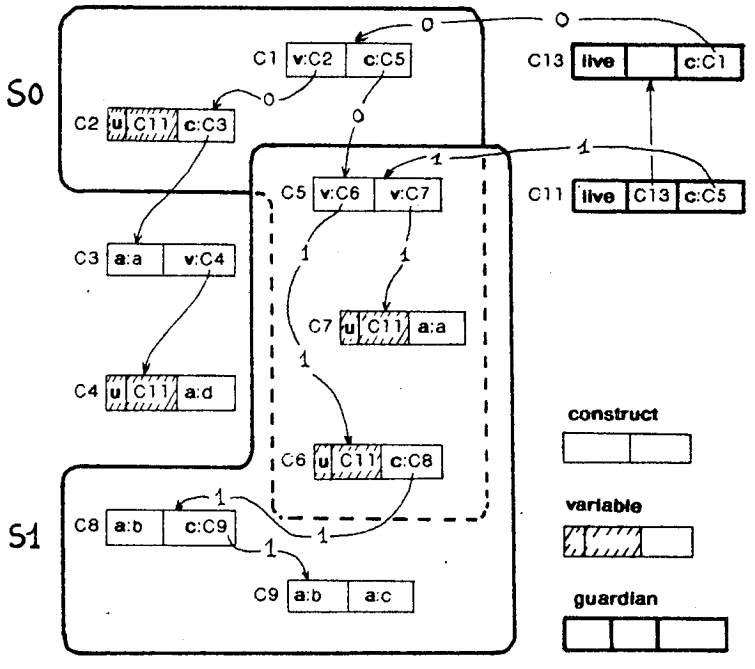


figure 2 – Walking through cells

After a level has been marked, the procedure **down_one_level** marks the corresponding guardian. Thus, comparing the level of the bindings with the level under marking simply amounts to test the allocation mark of the guardians.

5.2 Incidence of parallelism on marking

The execution of **retrieve** by the user process may lead to the logical destruction of the level currently under marking. In this case, the marking is aborted and the garbage-collector starts to mark the level below. Aborting a marking is possible if cells already marked by the collector do not give access to unmarked cell. To insure this condition, the cells get marked from the leaves to the root.

The parallel execution of the user process does not affect the principle of considering encountered marked cells as leaves. Indeed, cells allocated since the beginning of the current batch lead to cells which were allocated either during this batch, in which case they were marked at allocation time, or before this batch, in which case they can be accessed from the root given to the garbage-collector at batch start, and will be marked when encountered during the walk from this root.

6. Cell allocation, batches

The array **alloc_status** keeps a current allocation status for every cell. There are three status possibilities: **available**, qualifying an available cell, and the two batch indices 0 and 1, qualifying an allocated cell. The **current_batch** register contains the indice of the current batch. This indice is used to mark the cells at the time they are allocated, or when they are encountered by the collector in its marking phase.

Batches

Batches are introduced in the memory management to cope for parallelism between the user and the garbage-collector. At the beginning of the *i*-th batch, the allocation status of all non-available cells contains *i* mod 2. The *i*-th batch is associated with the indice (*i*+1) mod 2. During a batch, its indice is written in the allocation status of all cells undergoing allocation or encountered during the marking phase of the garbage-collector . After the marking phase, the cells which still have allocation status *i* mod 2 of the previous batch can be made available. This is done by the collecting phase, after which a new batch can be started.

```
current_batch :{0,1}
garbage_collector_idle :{true,false}

procedure wait_next_batch exclusion batches
| garbage_collector_idle:=true; suspend garbage_collector

procedure start_batch(n:name) exclusion batches
| if garbage_collector_idle then
| | current_batch:=(current_batch+1) mod 2
| | marking_level:=user_level; marking_name:=n
| | garbage_collector_idle:=false; restart garbage_collector
```

```
procedure cell_marking(ref:reference)
| status_alloc[ref]:=current_batch

procedure test_mark(ref:reference) :{marked,unmarked}
| if status_alloc[ref]=current_batch then result marked
| else result unmarked


available_cells :reference
status_alloc :array 1..maxref of {0,1,available}

procedure cell_allocation :reference exclusion allocation
| if available_cells=null ref then wait cell_available
| cell_marking(available_cells); result available_cells
| available_cells:=↑available_cells.next

procedure make_cell_available(ref:reference) exclusion allocation
| if status_alloc[ref]=(current_batch+1)mod 2 then
| | status_alloc[ref]:=available
| | ↑ref.next:=available_cells; available_cells:=ref
| | signal cell_available
```

7. Extensions

The machine presented has been simplified in order to exhibit the original aspects of a garbage-collector taking advantage of the memory usage stratification due to indeterminism. In the real machine, the algorithm are extended to treat the "cut" PROLOG primitive, which logicaly kills saved backtrack points and is a major cause of memory access loss, leading to full valorisation of the garbage-collector.

A simulator of the intermediate machine and a PROLOG interpreter using it are currently under development. A hardware realisation with two processors is under study. One of them will be microprogrammed to support the major part of the intermediate machine.

References

[1] "Implementing Prolog-compiling logic programs", D.H.D. Warren, D.A.I. Research Report, No. 39 and 40, University of Edinburgh, 1977.

[2] "List Processing in Real Time on a Serial Computer", H.G. Baker, Communications of the ACM, Vol. 21 No. 4 280-294, April 1978.

[3] "On-the-Fly Garbage Collection: An Exercise in Cooperation", E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, Communications of the ACM, Vol. 21 No. 11 966-975, Nov. 1978.

[4] "The memory management of PROLOG implementations", M. Bruynooghe, in logic programming eds Tarnlund and Clark, Academic press 1981.

[5] "Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques" Version 1, Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro, publication interne IRISA No 222, January 1984.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique