Une introduction à Prolog III

Alain Colmerauer
Professeur à l'Université Aix-Marseille II
Groupe Intelligence Artificielle
Unité de recherche Associée au CNRS 816
Faculté des Sciences de Luminy, Case 901
13288 Marseille Cedex 9

Résumé. Le langage de programmation Prolog III est une extension de Prolog au niveau de ce qu'il a de plus fondamental, le mécanisme d'unification. Il intègre dans ce mécanisme un traitement plus complet des arbres et des listes, un traitement numérique et un traitement de l'algèbre de Boole à deux valeurs. Nous présentons ici les spécifications de ce nouveau langage et illustrons ses possibilités au moyen d'exemples variés. Nous présentons aussi le modèle théorique de Prolog III qui, en fait, s'applique à toute une famille de langages de programmation. L'idée essentielle est de remplacer la notion d'unification par celle de résolution de contraintes.

Abstract. The Prolog III programming language extends Prolog by redefining the fundamental process at its heart : unification. Into this mechanism, Prolog III integrates refined processing of trees and lists, number processing, and processing of two-valued Boolean algebra. We present the specification of this new language and illustrate its capabilities by means of varied examples. We also present the theoretical foundations of Prolog III, which in fact apply to a whole family of programming languages. The central inovation is to replace the concept of unification by the concept of constraint solving.

vendredi, mars 24, 2000

INTRODUCTION

Le langage de programmation Prolog a été initialement conçu pour le traitement des langues naturelles. Son utilisation progressive pour résoudre des problèmes dans des domaines de plus en plus variés a mis en valeur ses qualités mais a aussi fait apparaître ses limites. Une partie de ces limites a été contournée par des implantations de plus en plus efficaces et des environnements de plus en plus riches. Il reste que le noyau même de Prolog, l'algorithme d'unification de Alan Robinson [22], est resté identique depuis quinze ans, et que ce noyau a perdu de son importance face à un ensemble toujours croissant de procédures externes prédéfinies. Citons par exemple les procédures nécessaires aux traitements numériques. Ces procédures externes sont d'un emploi difficile. Pour les appeler il faut être sûr que certains paramètres soient parfaitement connus, et ceci se heurte à la philosophie générale « prologienne » qui est de pouvoir parler n'importe où et n'importe quand d'un objet inconnu *x*.

Afin de remédier à cet état de choses nous avons profondément remanié Prolog en intégrant au niveau de l'unification: (1) une manipulation plus fine des arbres, qui peuvent être infinis, avec un traitement spécifique pour les listes, (2) un traitement complet de l'algèbre de Boole à deux valeurs, (3) un traitement numérique, comprenant l'addition, la soustraction, la multiplication par une constante et les relations <, =, =, >, (4) un traitement général de la relation ?. Ce remaniement s'est fait en remplaçant le concept même d'unification par celui de résolution de contraintes dans une structure mathématique donnée une bonne fois pour toutes. Par structure mathématique nous entendons ici un domaine muni d'opérations et de relations, les opérations n'étant pas forcément définies partout.

L'objet de cet article[1] est de présenter Prolog III, le nouveau langage ainsi obtenu, d'établir ses fondements et d'illustrer ses possibilités par des exemples intéressants. Ces fondements, qui s'appliquent en fait à toute une classe de langages de programmation « de type Prolog III », seront présentés à l'aide de notions mathématiques simples, sans faire explicitement appel à la logique du premier ordre.

Les travaux sur Prolog III ne sont pas isolés et d'autres recherches ont abouti à la conception de langages ayant avec lui des points communs. Citons le langage CLP(R) développé par J. Jaffar et S. Michaylov [19] dans lequel l'emphase est mise sur le traitement des nombres réels et le langage CHIP développé par l'équipe de M. Dincbas

---

[1] Une version trés préliminaire de cet article est parue dans *Proceedings of the 4th Annual ESPRIT Conference*, Bruxelles, North-Holland, pp. 611-629, septembre 1987.

et indépendant de la nature de l'étiquette attachée au nœud. L'ensemble des nœuds de l'arbre, par contre, peut être infini. Nous ne distinguons pas les arbre ayant un seul nœud des étiquettes que portent ces nœuds. Les identificateurs, les caractères, les valeurs booléennes, les nombres réels et les signes $<>^{\alpha}$ sont ainsi considérés comme des cas particuliers d'arbres.

Par *nombres réels* nous entendons ici des nombres réels au sens mathématique et non des nombres à virgule flottante. Nous faisons appel à la partition des réels en deux grandes catégories, les nombres rationnels, qui sont représentables par des fractions (et dont les nombres entiers sont un cas particulier) et les nombres irrationnels (comme par exemple π et $\sqrt{2}$) qu'aucune fraction ne peut représenter. En fait la machine calculera uniquement avec des nombres rationnels et ceci est lié à une propriété essentielle des différentes contraintes que l'on peut poser en Prolog III : si une variable est suffisamment contrainte pour ne représenter plus qu'un seul nombre réel alors ce nombre est forcément rationnel.

Un arbre *a* dont le nœud initial est étiqueté par le signe $<>^{\alpha}$ est appelé *liste* et est noté

$$<a_1,...,a_n>^{\alpha},$$

où $a_1...a_n$ est la suite (éventuellement vide) d'arbres qui constitue les fils immédiats de *a*. On se permet d'omettre α lorsque α est nul. Les *vraies* listes sont celles pour lesquels α est nul : elles servent à représenter des suites d'arbres (la suite de leurs fils immédiats). Les listes pour lesquels α n'est pas nul sont des listes *parasitaires* (qu'on n'a pu empécher d'exister) : elles représentent des suites d'arbres (la suite de leurs fils immédiats) complétées à droite par quelque chose d'inconnu et de longueur α. La *longueur* $|a|$ de la liste *a* est alors le réel $n+\alpha$. Les vraies listes ont pour longueur un nombre entier non négatif et les listes parasitaires ont pour longueur un nombre irrationnel positif. La liste <> est la seule liste qui ait une longueur nulle, on l'appelle la *liste vide*. On définit l'opération de *concaténation* entre une vraie liste et une liste quelconque par l'égalité :

$$<a_1,...,a_m>^{0} \bullet <b_1,...,b_n>^{\alpha} = <a_1,...,a_m,b_1,...,b_n>^{\alpha}.$$

L'opération est associative, $(a \bullet a') \bullet b = a \bullet (a' \bullet b)$, et la liste vide y joue le rôle d'élément neutre, $a \bullet <> = a$ et $<> \bullet b = b$. On remarque que pour toute liste *b* il existe une et seule vraie liste *a* et un et un seul réel α tels que

$$b = a \bullet <>^{\alpha}.$$

Cette liste *a* est appelée *préfixe* de *b* et est notée $\in b \vee$.

Opérations

Constantes

id : $\Lambda \mapsto$ id,
'c' : $\Lambda \mapsto$ 'c',
0' : $\Lambda \mapsto$ 0',
1' : $\Lambda \mapsto$ 1',
q : $\Lambda \mapsto$ q,
$<>^0$: $\Lambda \mapsto <>$,
$c_1...c_m$: $\Lambda \mapsto$ "$c_1...c_m$".

Opérations booléennes

¬ : $b_1 \mapsto \neg b_1$,
: $b_1 b_2 \mapsto b_1 \ b_2$,
Δ : $b_1 b_2 \mapsto b_1 \Delta b_2$,
: $b_1 b_2 \mapsto b_1 \ b_2$,
+ : $b_1 b_2 \mapsto b_1 + b_2$.

Opérations numériques

$+^1$: $r_1 \mapsto +r_1$,
$-^1$: $r_1 \mapsto -r_1$,
$+^2$: $r_1 r_2 \mapsto r_1 + r_2$,
$-^2$: $r_1 r_2 \mapsto r_1 - r_2$,
q∞ : $r_1 \mapsto$ q∞r_1,
/q' : $r_1 \mapsto r_1$/q'.

Opérations concernant les listes

|| : $l_1 \mapsto |l_1|$,
$<,>^m$: $a_1...a_m \mapsto <a_1,...,a_m>$,
$a_1...a_n$• : $l_1 \mapsto <a_1,...,a_n> \bullet l_1$.

Opérations générales

$()^{n+2}$: $e_1 a_2...a_{n+2} \mapsto e_1(a_2,...,a_{n+2})$,
[] : $e_1 l_2 \mapsto e_1[l_2]$.

Ici *id* désigne un identificateur, *c* et c_i un caractère, *q* et *q'* des nombres rationnel représenté par des fraction (ou des entiers), avec *q'* astreint à ne pas être nul, *m* un entier positif, *n* un entier non négatif et a_i un arbre quelconque. Le résultat des différentes opérations n'est défini que si b_i est une valeur booléenne, r_i un nombre réel, l_i une liste et e_i une étiquette qui n'est pas de la forme $<>^\alpha$.

notation préfixée.

Relations unaires

id : a_1 : id,
char : a_1 : char,
bool : a_1 : bool,
num : a_1 : num,
irint : a_1 : irint,
list : a_1 : list,
leaf : a_1 : leaf.

Relations d'égalités

= : $a_1 = a_2$,
? : a_1 ? a_2.

Relations booléennes

: a_1 a_2.

Relations numériques

< : $a_1 < a_2$,
> : $a_1 > a_2$,
= : $a_1 = a_2$,
= : $a_1 = a_2$,

Opérations approchées

$/^3$: $a_3 \doteq a_1 \,;\, a_2$,
∞^{n+1} : $a_{n+1} \doteq a_1 \dot{\infty} \ldots \dot{\infty} a_n$,
$\bullet^{n+1}$: $a_{n+1} \doteq a_1 \dot{:} \ldots \dot{:} a_n$.

Ici n désigne un entier plus grand que 1 et a_i un arbre quelconque. Les relations id, char, bool, num, irint, list, leaf permettent d'exprimer qu'un arbre a_1 est un identificateur, un caractère, une valeur booléenne, un nombre réel, un nombre entier ou irrationnel, une liste, une étiquette qui n'est pas de la forme $<>^{\alpha}$. Les relations = et ? correspondent bien entendu à l'égalité et à la non-égalité entre arbres. Le couple d'arbres a_1a_2 n'est dans la relation que si a_1 et a_2 sont des valeurs booléennes et si $a_1 = 1'$ entraîne $a_2 = 1'$. Le couple d'arbres a_1a_2 n'est dans la relation <, >, =, = que s'il constitue un couple de réels qui est dans la relation classique correspondante.

Termes et contraintes

Plaçons-nous dans une structure (D, F, R) et, pour désigner les éléments de son domaine D, donnons-nous une bonne fois pour toutes un ensemble *universel* V de variables. Cet ensemble sera supposé infini et dénombrable. Nous pouvons maintenant construire des objets syntaxiques de deux sortes, les termes et les contraintes. Les *termes* sont des suites d'éléments juxtaposés de V≈F de l'une des deux formes,

$$x \quad \text{ou} \quad f\, t_1 ... t_n,$$

où x est une variable, f est une opération à n positions et les t_i des termes plus simples. Les *contraintes* sont des suite d'éléments juxtaposés de V≈F≈R de la forme,

$$r\, t_1 ... t_n,$$

où r est une relation à n positions et les t_i des termes. On remarquera que dans la définition des termes nous n'avons imposé aucune restriction sur la compatibilité sémantique des f avec les t_i. Ces restrictions, comme nous allons le voir, sont implicitement contenues dans le mécanisme qui permet de passer d'un terme à sa « valeur ».

On introduit tout d'abord la notion d'*affectation* σ d'un sous-ensemble W de variables : c'est tout simplement une application de W dans le domaine D de la structure. Cette application σ se prolonge naturellement en une application σ^* d'un ensemble T_σ de termes vers le domaine D, en considérant que

$$\sigma^*(x) = \sigma(x),$$
$$\sigma^*(f\, t_1 ... t_n) = f\, \sigma^*(t_1) ... \sigma^*(t_n).$$

Les termes qui ne font pas partie de T_σ sont ceux qui font intervenir des variables ne figurant pas dans W ou des opérations partielles f non définies sur les arguments $\sigma^*(t_i)$. Suivant qu'un terme t appartient ou n'appartient pas à T_σ sa *valeur* dans l'affectation σ est définie et égale à $\sigma^*(t)$ ou n'est pas définie. D'une façon intuitive la valeur d'un terme dans une affectation de variables est obtenue en remplaçant les variables par leurs valeurs et en évaluant le terme. Si cette évaluation ne peut être effectuée, la valeur du terme n'est pas définie pour l'affectation considérée.

Nous dirons que l'affectation σ d'un ensemble de variables *satisfait* la contrainte $r\, t_1 ... t_n$ si la valeur $\sigma^*(t_i)$ de chaque terme t_i est définie et si le n-uplet $\sigma^*(t_1) ... \sigma^*(t_n)$ est dans la relation r, c'est-à-dire si

sur W.

Voici dans notre structure quelques exemples pour illustrer toutes ces notions.
- L'affectation σ de V définie par σ(x) = 1', pour toute variable x, est une solution du système de contraintes {$x = y$, y ? 0} mais n'est pas une solution du système {$x = y$, $+y$? 0}.
- L'affectation σ de {y} définie par σ(y) = 4 est une solution sur {y} du système {$x = y$, y ? 0}.
- Les systèmes de contraintes {$x = y$, $+y$? 0} et {$–x = –y$, y ? 0} sont équivalents. De même le système {1 = 1, $x = x$} est équivalent au système vide de contraintes.
- Les systèmes {$x = y$, $y = z$, x ? z} et {$x < z$} ne sont pas équivalents, mais sont équivalents sur le sous-ensemble de variables {x, z}.

On notera que les systèmes solubles de contraintes sont tous équivalents sur l'ensemble vide de variables et que les systèmes non solubles sont tous équivalents. Par système *soluble* on entend un système ayant au moins une solution.

La première chose que Prolog III permet de faire est de déterminer si un système de contraintes est soluble et dans l'affirmative de le résoudre. Par exemple pour connaître le nombre x de pigeons et le nombre y de lapins qui ensemble totalisent 12 têtes et 34 pattes il suffira de poser la requête

{$x+y$ = 12, $2x+4y$ = 34} ?

et la machine répondra

{x = 7, y = 5}.

Pour connaître la suite z de 10 éléments qui produit la même suite si on lui concatène à gauche la suite 1,2,3 ou à droite la suite 2,3,1 il suffira de poser la requête

{|z| = 10, <1,2,3>•z ≐ z• <2,3,1>} ?

pour obtenir l'unique réponse

= <1,2,3,1,2,3,1,2,3,1>}.

Si dans la requête on remplace la liste <2,3,1> par la liste <2,1,3> on n'obtient aucune réponse, ce qui signifie que le système

{|z| = 10, <1,2,3>•z ≐ z• <2,1,3>}

d = 1' pour « quelque chose existe par la nécessité de sa propre nature »,
e = 1' pour « quelque chose existe par la volonté d'un autre être ».
Les 5 prémisses se traduisent alors par le système de contraintes

$$\{a = 1',\ a \quad b\Delta c,\ a \quad d\Delta e,\ d \quad b,\ e \quad \neg c\}$$

qui posé comme requête donne la réponse

$$\{a = 1',\ b = 1',\ d\Delta e = 1',\ e\Delta c = 1'\}.$$

On constate que b est bien contraint à valoir 1'.

Après ces exemples il est nécessaire de préciser ce que nous entendons par « résoudre » un système S de contraintes faisant intervenir un ensemble W de variables. Intuitivement il s'agit de trouver toutes les solutions de S sur W. Comme il peut avoir une infinité de telles solutions il n'est pas possible de les énumérer. Ce qui est possible par contre est de calculer un système sous forme « résolue » équivalent à S sur W et dont les solutions « les plus intéressantes » sont apparentes. Plus précisément par système sous forme *résolue* nous entendons un système soluble dans lequel, pour chaque variable x, la solution de S sur $\{x\}$, quand elle est unique, est apparente. Le lecteur vérifiera que dans les exemples précédents les systèmes fournis en réponse étaient bien sous forme résolue.

Avant de terminer cette partie citons une propriété essentielle qui facilite la résolution des systèmes de contraintes dans la structure choisie.

PROPRIETE. Si S est un système de contraintes Prolog III et W un ensemble de variables, alors les deux propositions suivantes sont équivalentes :
(1) pour chaque x de W, il y a plusieurs solutions numériques de S sur $\{x\}$;
(2) il existe une solution numérique irrationnelle de S sur W.

Par solution numérique, ou solution numérique irrationnelle, sur un ensemble de variables, nous entendons ici une solution dans laquelle toutes les variables de cet ensemble prennent pour valeurs des nombres réels, ou prennent pour valeurs des nombres irrationnels.

SEMANTIQUE DES LANGAGES DE TYPE PROLOG III

A partir de la structure que nous avons choisie nous pouvons maintenant définir le langage de programmation Prolog III. La façon de procéder étant indépendante de la structure choisie, nous définirons en fait ce qu'est le langage de « type Prolog III » associé à une structure donnée. La seule supposition que nous ferons est que la relation d'égalité figure dans l'ensemble des relations de la structure considérée.

RepasLéger(h, p, d) →
HorsDœuvre(h, i)
Plat(p, j)
Dessert(d, k),
{$i = 0$, $j = 0$, $k = 0$, $i+j+k = 10$};

Plat(p, i) → Viande(p, i);
Plat(p, i) → Poisson(p, i);

HorsDœuvre(radis, 1) →;
HorsDœuvre(pâté, 6) →;

Viande(bœuf, 5) →;
Viande(porc, 7) →;

Poisson(sole, 2) →;
Poisson(thon, 4) →;

Dessert(fruit, 2) →;
Dessert(glace, 6) →;

La première règle se lit : « sous réserve que les quatre conditions $i = 0$, $j = 0$, $k = 0$, $i+j+k = 10$ soient satisfaites, le triplet h,p,d constitue un repas léger si h est un hors-d'œuvre de valeur calorique i, si t est un plat de valeur calorique j et si d est un dessert de valeur calorique k ». La dernière règle se lit : « la glace est un dessert de valeur calorique 6 ».

Donnons maintenant une définition précise de l'ensemble des éléments acceptables. Les règles du programme sont en fait des schémas de règles. Chaque règle (de la forme précédemment mentionnée) remplace l'ensemble des *règles évaluées*

$$\sigma^*(t_0) \rightarrow \sigma^*(t_1)...\sigma^*(t_n)$$

obtenues en considérant toutes les solutions σ de S pour lesquels les valeurs $\sigma^*(t_i)$ sont définies. Chaque règle évaluée

$$a_0 \rightarrow a_1...a_n,$$

qui porte en fait uniquement sur des éléments a_i du domaine, peut s'interpréter de deux façons:

{HorsDœuvre(pâté,6), Poisson(sole,2), Dessert(fruit,2)},
{Plat(sole,2)},
{RepasLéger(pâté,sole,fruit)}

et donc que l'arbre

RepasLéger(pâté,sole,fruit)

est acceptable. Si l'on considère ces règles comme des règles de réécriture, la suite constituée uniquement de ce dernier arbre peut être effacée par la suite de réécritures,

RepasLéger(pâté,sole,fruit) →
HorsDœuvre(pâté,6) Plat(sole,2) Dessert(fruit,2) →
Plat(sole,2) Dessert(fruit,2) →
Poisson(sole,2) Dessert(fruit,2) →
Dessert(fruit,2) →,

ce qui confirme bien qu'il s'agissait d'un élément acceptable.

Exécution d'un programme

Nous venons de montrer quelle est l'information implicite contenue dans un programme écrit dans un langage de type Prolog III, mais nous n'avons pas montré en quoi consiste l'exécution d'un tel programme. Cette exécution vise à résoudre le problème suivant : « étant donnés une suite $t_1...t_n$ de termes et un système S de contraintes, le tout faisant intervenir un ensemble W de variables, trouver les solutions de S sur W pour lesquelles les valeurs des termes t_i sont toutes des éléments acceptables du domaine ». Ce problème sera soumis à la machine en posant la *requête*

$t_1.... t_n$, S ?

Deux cas particuliers retiendront notre attention. (1) Si la suite $t_1...t_n$ est vide alors la requête se résume à demander si le système S est soluble et dans l'affirmative de le résoudre le système S. Nous avons déjà donné de tels exemples de requêtes. (2) Si le système S est vide (ou absent) et que la suite de termes est réduite à un seul terme alors la requête se résume à: « quelles sont les valeurs des variables qui transforment ce terme en un élément acceptable ?». Ainsi si l'on se reporte à l'exemple précédent de programme, la requête

RepasLéger(*h*, *p*, *d*) ?

Reconsidérons notre premier exemple de programme et appliquons ce traitement à la requête

RepasLéger(h, p, d) ?

L'état initial de la machine est
($\{h,p,d\}$, RepasLéger(h,p,d), $\{\}$).
En appliquant la règle
RepasLéger(h', p', d') → HorsDœuvre(h', i) Plat(p', j) Dessert(d', k),
$\{i = 0, j = 0, k = 0, i+j+k = 10\}$
on passe à l'état
($\{h,p,d\}$, HorsDœuvre(h',i) Plat(p',j) Dessert(d',k),
$\{i=0, j=0, k=0, i+j+k=10$, RepasLéger($h$,$p$,$d$)=RepasLéger($h'$,$p'$,$d'$)$\}$)
qui se simplifie en
($\{h,p,d\}$, HorsDœuvre(h',i) Plat(p',j) Dessert(d',k),
$\{i=0, j=0, k=0, i+j+k=10, h=h', p=p', d=d'\}$),
puis en
($\{h,p,d\}$, HorsDœuvre(h,i) Plat(p,j) Dessert(d,k),
$\{i=0, j=0, k=0, i+j+k=10\}$).
En appliquant la règle
HorsDœuvre(pâté, 6) →
et en simplifiant on passe à l'état
($\{h,p,d\}$, Plat(p,j) Dessert(d,k), $\{h$=pâté, j=0, k=0, j+k=4$\}$).
En appliquant la règle
Plat(p', i) → Poisson(p', i)
et en simplifiant un peu on passe à l'état
($\{h,p,d\}$, Poisson(p',i) Dessert(d,k),
$\{h$=pâté, j=0, k=0, j+k=4, $p=p'$, $j=i\}$).
qui se simplifie encore en
($\{h,p,d\}$, Poisson(p,j) Dessert(d,k), $\{h$=pâté, j=0, k=0, j+k=4$\}$).
En appliquant la règle
Poisson(sole, 2) →
on obtient
($\{h,p,d\}$, Dessert(d,k), $\{h$=pâté, p=sole, k=0, k=2$\}$).
Finalement en appliquant la règle
Dessert(fruit, 2) →
on obtient
($\{h,p,d\}$, $\wedge$, $\{h$=pâté, p=sole, d=fruit$\}$).
On en conclut que le système
$\{h$=pâté, p=sole, d=fruit$\}$
constitue une des réponses à la requête posée.

de programmes. Nous aborderons successivement le traitement des nombres, le traitement des booléens, le traitement des arbres et des listes et finalement le traitement des entiers.

Calcul de prêt

Il s'agit tout d'abord de calculer les suites possibles de versements à effectuer pour rembourser un capital emprunté à une banque. On suppose qu'entre deux versements s'écoule toujours le même temps et que pendant cette durée le taux d'intérêt réclamé par la banque est de 10%. Les arbres acceptables seront de la forme

VersementsCapital(*x*, *c*),

où *x* est la suite des versements nécessaires pour rembourser le capital *c* avec un intérêt de 10% entre deux versements. Le programme lui-même se résume à deux règles:

```
VersementsCapital(<>, 0) →;
VersementsCapital(<v>•x, c) →
        VersementsCapital(x, (110/100)c–v);
```

La première règle exprime qu'il n'est pas nécessaire de faire de versements pour rembourser un capital nul. La deuxième règle exprime que la suite des n+1 versements pour rembourser un capital c consiste en un versement v et d'un suite s de n versements permettant de rembourser le capital c augmenté de 10% d'intérêts mais le tout diminué du versement v effectué.

Ce programme peut être utilisé de différentes façons. Une des plus spectaculaires est de demander pour quelle valeur de v la suite de versements $<v,2v,3v>$ permet de rembourser 1000F. Il suffit de poser la requête

VersementsCapital(<*v*,2*v*,3*v*>, 1000) ?

pour obtenir la réponse

{*v* = 207 + 413/641}.

Voici une trace abrégée de ce calcul. On part de l'état initial

({*v*}, VersementsCapital(<*v*,2*v*,3*v*>,1000), {}).

En appliquant la règle

VersementsCapital(<*v'*>•*x*,*c*) → VersementsCapital(*x*,(110/100)*c*-*v'*)

on passe à l'état

({*v*}, VersementsCapital(*x*,(110/100)*c*-*v'*),

Calcul de la périodicité d'une suite

Ce problème a été proposé en [5]. On considère la suite infinie de nombres réels définie par

$$x_{i+2} = |x_{i+1}| - x_i$$

où x_1 et x_2 sont quelconques. Il s'agit de montrer que cette suite est toujours périodique et que sa période est de 9, c'est-à-dire que les suites

$$x_1, x_2, x_3, \ldots \quad \text{et} \quad x_{1+9}, x_{2+9}, x_{3+9}, \ldots$$

sont toujours identiques.

Chacune de ces deux suites est complètement déterminée si l'on fixe ses deux premiers éléments. Pour montrer qu'elles sont égales il suffira donc de montrer que dans toute suite de onze éléments

$$x_1, x_2, x_3, \ldots, x_{10}, x_{11}$$

on a

$$x_1 = x_{10} \quad \text{et} \quad x_2 = x_{11}.$$

Voici tout d'abord le programme qui énumère toutes les suites finies $x_1,x_2,\ldots,x_n$ respectant la loi donnée plus haut :

```
Suite(<+y, +x>) →;
Suite(<z,y,x>•s) →
    Suite(<y,x>•s)
    ValeurAbsolue(y, y'), {z = y'–x};

ValeurAbsolue(y, y) →, {y = 0};
ValeurAbsolue(y, –y) →, {y < 0};
```

Les signes + de la première règle imposent que x et y désignent des nombres. On remarquera que les suites sont énumérées de gauche à droite, c'est-à-dire que les arbres de la forme Suite(s) ne sont acceptables que si s est de la forme $<x_n,\ldots,x_2,x_1>$. Si on lance ce programme en posant la requête

$$\text{Suite}(s),\ \{|s| = 11,\ s \doteq w{:}\ v{:}\ u,\ |u| = 2,\ |w| = 2,\ u\ ?\ w\}\ ?$$

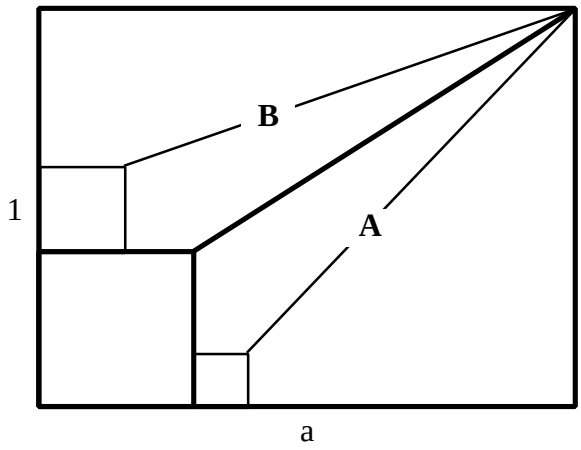
On désignera par *a* le rapport de la longueur du plus grand côté avec celle du plus petit côté du rectangle construit. On peut évidemment supposer que la longueur du plus petit côté est 1 et que la longueur du plus grand côté est *a*. Il faut donc remplir un rectangle de dimensions 1∞*a* par *n* carrés tous distincts. En se référant au schéma qui suit, la base de l'algorithme de remplissage consistera

(1) à placer un carré dans le coin inférieur gauche du rectangle,

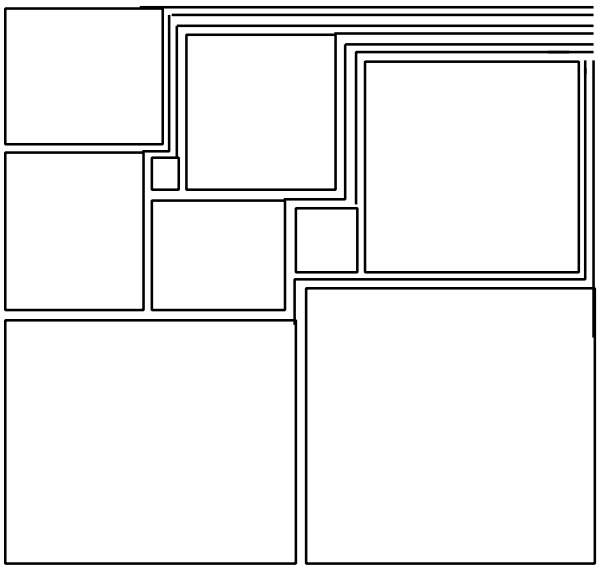
(2) à remplir de carrés la zone A, si elle n'est pas vide,

(3) à remplir de carrés la zone B, si elle n'est pas vide.

Le remplissage des zones A et B se fera récursivement de la même façon : placer un carré dans le coin inférieur gauche et remplir deux sous-zones.

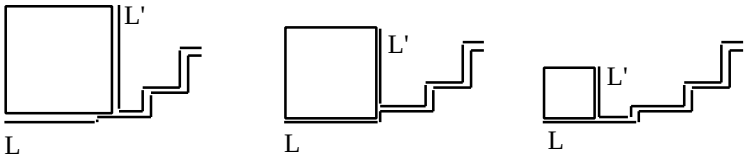


Les zones et les sous-zones sont séparées par des lignes brisées en forme d'escalier allant du coin supérieur droit des carrés au coin supérieur droit du rectangle. Ces lignes brisées ne descendent jamais et s'il est possible d'en tracer plusieurs pour aller d'un point à un autre on considère toujours la plus basse. Voici par exemple toutes les lignes de séparations correspondant à la première solution du problème lorsque *n* = 9 :



CarréPlacé(*b*, *L*, *L'*),

et ne les accepter que s'il est possible de placer un carré de dimension *b*∞*b* sur le début de la ligne *L* et que si *L'* est la ligne constituée du coté vertical droit de ce carré prolongée par la partie droite de la ligne *L* (voir figure ci-dessous). En fait *L* désigne la ligne inférieure d'une zone mais de laquelle on a enlevé le premier segment vertical. La figure ci-dessous montre les trois cas qui peuvent se présenter et qui se matérialiseront par trois règles. Soit le carré déborde sur la première marche, qui en fait était une fausse marche de hauteur nulle, soit le carré est collé contre la première marche, soit le carré n'est pas assez grand pour toucher la première marche.



Le programme lui même est constitué des dix règles suivantes :

première règle, qui à la fois, contraint a à être plus grand ou égal à 1, crée n carrés (de tailles inconnues) tous distincts et lance le remplissage de la zone constituée par la totalité du rectangle. La ligne L constituant la délimitation supérieure de cette zone est inconnue au départ, mais, compte tenu qu'elle doit joindre sans descendre deux points qui se trouvent à la même hauteur, ce sera forcément une ligne horizontale (représentée par un escalier dont toutes les marches sont de hauteur nulle). Si on lance la requête

RectangleRempli(a, C), {$|C| = 9$} ?

on obtient 8 réponses. Les deux premières que voici

{a = 33/32, C = <15/32, 9/16, 1/4, 7/32, 1/8, 7/16, 1/32, 5/16, 9/32>},

{a = 69/61, C = <33/61, 36/61, 28/61, 5/61, 2/61, 9/61, 25/61, 7/61, 16/61>}.

correspondent aux deux assemblages que nous avons dessinés. Les 6 autres réponses décrivent des assemblages symétriques de ceux-ci. Pour retrouver les positions des différents carrés dans le rectangle on peut procéder ainsi. On remplit le rectangle en utilisant successivement chaque carré de la liste C dans son ordre d'apparition. A chaque étape on considère tous les coins libres qui ont la même orientation que le coin gauche inférieur du rectangle et on choisit celui qui se trouve le plus à droite pour y placer le carré.

Il existe une littérature importante autour du problème que nous venons de traiter. Signalons deux résultats importants. Il a été montré en [25] que quel que soit le nombre rationnel a = 1 il existe toujours un entier n tel que le rectangle de dimension 1∞a puisse être rempli par n carrés de tailles distinctes. Dans le cas ou a = 1, c'est-à-dire lorsque le rectangle à remplir est un carré, il a été montré en [14] que le plus petit n possible est n = 21.

TRAITEMENT DES BOOLEENS

Calcul de pannes

Ici il s'agit de détecter le ou les composants défectueux dans un additionneur qui calcule la somme binaire de trois bits x_1, x_2, x_3 sous forme d'un nombre binaire de deux bits y_1y_2. Comme on peut le voir ce circuit proposé en [15] est formé de 5 composants numérotés de 1 à 5: deux portes *et* (marquées Et), une porte *ou* (marquée Ou) et deux portes *ou exclusif* (marquées OuX). On a aussi introduit trois variables u_1, u_2, u_3 pour représenter les sorties des portes 1, 2 et 4.

Si l'état du circuit nous amène à poser la requête

$$\text{Circuit}(<1', 1', 0'>, <0', 1'>, <p1, p2, p3, p4, p5>) ?$$

on diagnostique que le composant numéro 4 est en panne,

$$\{ p1= 0',\ p2 = 0',\ p3 = 0', p4 = 1', p5 = 0' \}.$$

Si l'état du circuit nous amène à poser la requête

$$\text{Circuit}(<1', 0', 1'>, <0', 0'>, <p1, p2, p3, p4, p5>)?$$

on diagnostique que soit le composant numéro 1 soit le composant numéro 3 est en panne,

$$\{ p1 \Delta p3 = 1', p1\ \ p3 = 0', p2 = 0',\ p4 = 0', p5 = 0'\}.$$

Raisonnement logique

On considère maintenant les 18 phrases d'un puzzle de Lewis Carroll [7] que nous reproduisons ci-dessous. Il s'agit de répondre à des questions du genre : « quel lien existe-t-il entre avoir l'esprit clair, être populaire et être apte à être député ? » ou « quel lien existe-t-il entre savoir garder un secret, être apte à être député et valoir son pesant d'or ?».

1. Tout individu apte à être député et qui ne passe pas son temps à faire des discours, est un bienfaiteur du peuple.

2. Les gens à l'esprit clair, et qui s'expriment bien, ont reçu une éducation convenable.

3. Une femme qui est digne d'éloges est une femme qui sait garder un secret.

4. Les gens qui rendent des services au peuple, mais n'emploient pas leur influence à des fins méritoires, ne sont pas aptes à être députés.

5. Les gens qui valent leur pesant d'or et qui sont dignes d'éloges, sont toujours sans prétention.

6. Les bienfaiteurs du peuple qui emploient leur influence à des fins méritoires sont dignes d'éloges.

7. Les gens qui sont impopulaires et qui ne valent pas leur pesant d'or, ne savent pas garder un secret.

8. Les gens qui savent parler pendant des heures et des heures et qui sont aptes à être députés, sont dignes d'éloges.

9. Tout individu qui sait garder un secret et qui est sans prétention, est un bienfaiteur du peuple dont le souvenir restera impérissable.

10. Une femme qui rend des services au peuple est toujours populaire.

Possibilité(<
<*a*,"avoir l'esprit clair">,
<*b*,"avoir reçu une bonne éducation">,
<*c*,"discourir sans cesse">,
<*d*,"employer son influence à des fins méritoires">,
<*e*,"être affiché dans les vitrines">,
<*f*,"être apte à être député">,
<*g*,"être un bienfaiteur du peuple">,
<*h*,"être digne d'éloges">,
<*i*,"être populaire">,
<*j*,"être sans prétention">,
<*k*,"être une femme">,
<*l*,"laisser un souvenir impérissable">,
<*m*,"posséder une influence">,
<*n*,"savoir garder un secret">,
<*o*,"s'exprimer bien">,
<*p*,"valoir son pesant d'or">>) $\rightarrow$,

{(*f* ¬*c*) *g*,
(*a* *o*) *b*,
(*k* *h*) *n*,
(*g* ¬*d*) ¬*f*,
(*p* *h*) *j*,
(*g* *d*) *h*,
(¬*i* ¬*p*) ¬*n*,
(*c* *f*) *h*,
(*n* *j*) (*g* *l*),
(*k* *g*) *i*,
(*p* *c* *l*) *e*,
(*k* ¬*a* ¬ *b*) ¬*f*,
(*n* ¬*c*) ¬*i*,
(*a* *m* *d*) *g*,
(*g* *j*) ¬*e*,
(*n* *d*) *p*,
(¬*o* ¬*m*) ¬*k*,
(*i* *h*) (*g* Δ *j*)};

Pour extraire une partie de l'information on ajoute les règles :

ce qui veut dire que les gens qui savent garder un secret et sont aptes à être député valent leur pesant d'or.

En fait dans ces deux exemples d'éxecution on a supposé que Prolog III produisait comme réponse des systèmes résolus très simplifiés et notamment ne contenant pas de variables booléennes inutiles. Si ce n'était pas le cas, pour montrer (et non pas trouver) que les gens qui savent garder un secret et son apte à être député valent leur pesant d'or il aurait fallu poser la requête

```
SousPossibilité(<
        <p,"savoir garder un secret">,
        <q,"être apte à être député">,
        <r,"valoir son pesant d'or">>),
{x = (p   q    r)} ?
```

et obtenir une réponse de la forme {*x* = 1', ... } ou encore n'obtenir aucune réponse à la requête

```
SousPossibilité(<
        <p,"savoir garder un secret">,
        <q,"être apte à être député">,
        <r,"valoir son pesant d'or">>),
{(p   q    r) = 0'} ?
```

TRAITEMENT DES ARBRES ET DES LISTES

Calcul des feuilles d'un arbre

Voici tout d'abord un exemple se servant de la possibilité, offerte par l'opération [], d'accéder aux étiquettes et aux fils d'un arbre. Il s'agit de calculer la liste des feuilles d'un arbre fini en omettant les feuilles étiquetées <>α. Voici le programme :

Comme réponse à la requête

Valeur(<1,9,9,0>, *x*) ?

on obtient

{x = 1990}.

Calcul de la liste inversée

Si l'on sait accéder au premier et au dernier élément d'une liste, il doit être possible d'écrire un joli programme pour inverser une liste. Voici celui que je propose :

```
Inversion(x, y) →
        Palindrome(u),
        {u ≐ x• y, |x| = |y|};

Palindrome(<>) → ;
Palindrome(v) →
        Palindrome(u),
        {v ≐ <a>• u• <a>};
```

Chacune des deux requêtes

Inversion(<1,2,3,4,5>, *x*) ?
Inversion(*x*, <1,2,3,4,5>) ?

produit la même réponse

{*x* = <5,4,3,2,1>}.

Pour la requête

Inversion(*x*, *y*) Inversion(*y*, *z*), {*x* ? *z*, |*x*| = 10} ?

on n'obtient aucune réponse, ce qui confirme qu'une liste retournée deux fois est égale à elle-même.

FormeS("aaabbbb") ?

ne produit aucune réponse ce qui signifie que la chaîne "aaabbbb" ne fait pas partie du langage.

TRAITEMENT DES ENTIERS

Les algorithmes pour résoudre des contraintes sur les nombres entiers sont complexes et souvent inefficaces. C'est pour cette raison que la structure sous-jacente à Prolog III ne comporte pas de relation permettant de contraindre un nombre à n'être qu'un entier. Nous avons cependant prévu une fonctionnalité permettant d'énumérer des entiers satisfaisant à l'ensemble courant de contraintes.

Enumération d'entiers

On modifie le comportement de la machine abstraite Prolog III de façon à ce qu'elle se comporte comme si l'ensemble infini de règles

enum(0) →;
enum(−1) →;
enum(1) →;
enum(−2) →;
enum(2) →;
.....

était ajouté à tout programme. De plus on implante la machine abstraite de façon à ce que la recherche de l'ensemble des règles applicables se passe en un temps fini chaque fois que cet ensemble est fini. Si l'on se reporte à la définition de la machine abstraite ceci revient à lui ajouter toutes les transitions de la forme

$$(W, t_0\ t_1 ... t_m\ , S) \ \rightarrow \ (W, t_1 ... t_m\ , S \approx \{t_0 = \text{enum}(\mathbf{n})\}),$$

où **n** est un entier tel que le système $S \approx \{t_0 = \text{enum}(\mathbf{n})\}$ admette au moins une solution dans laquelle les valeurs des t_i sont toutes définies.

Par exemple, si dans l'état courant de la machine abstraite le premier terme à effacer est « enum(x) » et que le système courant de contrainte S est équivalent sur $\{x\}$ à $\{3/4 = x, x = 3+1/4\}$, il y aura deux transitions : l'une vers un état avec un système équivalent à $S \approx \{x=1\}$, l'autre vers un état avec un système équivalent à $S \approx \{x=2\}$.

Voici un autre problème pour illustrer l'énumération d'entiers. Il s'agit de résoudre un puzzle classique de crypto-arithmétique : affecter les dix chiffres 0,1,2,3,4,5,6,7,8,9 aux dix lettres *D,G,R,O,E,N,B,A,L,T* de façon à ce que l'addition *DONALD* + *GERALD* = *ROBERT* soit juste. On met en place d'une façon déterministe le maximum de contraintes numériques portant sur des réels et on utilise le non déterminisme pour énumérer tous les entiers qui doivent satisfaire à ces contraintes. Voici le programme sans commentaires :

```
Solution(i, j, i+j) →
        Valeur(<D,O,N,A,L,D>, i)
        Valeur(<G,E,R,A,L,D>, j)
        Valeur(<R,O,B,E,R,T>, i+j)
        DifférentsEtEntre09(x)
        Entiers(x),
        {<D,G,R,E,N,B,A,L,T,O> = x,
        D ? 0, G ? 0, R ? 0};

Valeur(<>, 0)→ ;
Valeur(y, 10i+j) →
        Valeur(x, i), {y ≐ x• <j>};

DifférentsEtEntre09(<>) →;
DifférentsEtEntre09(<i>•x) →
        HorsDe(i, x)
        DifférentsEtEntre09(x),
        {0 = i, i = 9};

HorsDe(i, <>) →;
HorsDe(i, <j>•x) →
        HorsDe(i, x), {i ? j};

Entiers(<>) →;
Entiers(<i>•x) →
        enum(i) Entiers(x);
```

La réponse à la requête

Solution(*i, j, k*) ?

est

La deuxième est que

$$0x_1+1x_2+ \ldots +(n-1)x_n = n(n+1)/2,$$

car la somme des nombres qui apparaissent dans la phrase est à la fois $1x_1+2x_2+\ldots+nx_n$ et $x_1+\ldots+x_n + 1+\ldots+n$. De tout ces considérations il résulte le programme final suivant :

```
Solution(X) →
        Somme(X, 2n)
        SommePondérée(X, m)
        Décompte(X, X)
        Entiers(X),
        {n = |X|, m ≐ n∻(n+1) /2};

Somme(<>, 0) →;
Somme(<x>•X, x+y) →
        Somme(X, y);

SommePondérée(<>, 0) →;
SommePondérée(X•<x>, z+y) →
        SommePondérée(X, y),
        {z ≐ |X|∻ x};

Décompte(<>, Y) →,
        {<1>•Y ≐ Y:• <1>};
Décompte(<x>•X, Y') →
        Décompte(X, Y),
        {Y' ≐ U:• <y+1>:• V,
        Y ≐ U:• <y>:• V,
        |U | = x–1};

Entiers(<>) →;
Entiers(<x>•X) →
        Entiers(X)
        enum(x);
```

En donnant successivement à n les valeurs 1, 2, ... , 20 et en posant la requête

Solution(X), { $|X| = n$} ?

Repas légers	4 sec
Calcul de prêt, $n = 3$	2 sec
Calcul de prêt, $n = 50$	6 sec
Calcul de prêt, $n = 100$	23 sec
Suite périodique	3 sec
Carrés, $n = 9$	13 min 15 sec
Carrés, $n = 9$, 1ère solution	1 min 21 sec
Carrés, $n = 10$, 1ère solution	6 min 36 sec
Carrés, $n = 11$, 1ère solution	1 min 38 sec
Carrés, $n = 12$, 1ère solution	5 min 02 sec
Carrés, $n = 13$, 1ère solution	4 min 17 sec
Carrés, $n = 14$, 1ère solution	13 min 05 sec
Carrés, $n = 15$, 1ère solution	11 min 29 sec
Pannes, 2ème requête	3 sec
Lewis Carrol, 2ème requête	3 sec
Donald + Gerald ...	1 min 33 sec
Auto-référence, $n = 4$	3 sec
Auto-référence, $n = 5$	4 sec
Auto-référence, $n = 10$	11 sec
Auto-référence, $n = 15$	36 sec
Auto-référence, $n = 20$	1 min 54 sec
Auto-référence, $n = 25$	5 min 51 sec
Auto-référence, $n = 30$	17 min 55 sec

Tous les temps, sauf indication contraire, sont des temps d'exécution d'un programme complet (y compris le temps de lecture et d'écriture des requêtes et des réponses). Le calcul de prêt consiste à calculer la suite de versements v, $2v$, $3v$, ... ,nv qui permet de rembourser un capital de 1000. Pour mieux juger des résultats il faut prendre en compte le fait que tous les calculs sont faits en précision infinie. Pour le calcul de prêt avec $n = 100$ on produit une fraction simplifiée ayant un numérateur et un dénominateur de plus de 100 chiffres!

Terminons cet article par quelques informations sur l'implémentation de Prolog III. Le noyau de l'interpréteur Prolog III consiste en une machine à deux piles qui explore par « backtracking » l'espace de recherche de la machine abstraite. Ces deux piles se remplissent et se vident conjointement. Dans la première on crée les structures qui représentent les états à travers lesquels on passe. Dans la deuxième on note par des couples adresse-valeur toutes les modifications que l'on fait sur la première pile et ceci afin de pouvoir faire les restaurations nécessaires lors d'un « backtrack ». Un système général de récupération de mémoire [23] permet de déceler les informations devenues inaccessibles et de récupérer la place qu'elles occupent en retassant les deux piles. Lors de ce retassement la topographie

Le prototype a été réalisé conjointement par notre laboratoire (le GIA) et la société PrologIA. Des aides financières importantes ont été obtenues d'une part du Centre National d'Etudes des Télécommunications (marché 86 1B 027) et d'autre part des Communautés Européennes, dans le cadre du projet ESPRIT P1106 « Further development of Prolog and its Validation by KBS in Technical Areas » avec pour autres partenaires les sociétés Daimler-Benz et Bosch. Des aides additionnelles ont été obtenues du CEA dans le cadre de l'Association Méditérannéenne pour le Développement de l'IA, de la compagnie DEC dans le cadre d'un « External Research Grant » et du Ministère de la Recherche et de l'Enseignement Supérieur dans le cadre des Programmes de Recherches Concertés « Génie Logiciel » et « Intelligence Artificielle ». Enfin les travaux plus récents sur la multiplication et la concaténation approchée ont été financés par les Communautés Européennes, dans le cadre du projet « Basic Research » intitulé « Computational Logic ».

Je remercie toute l'équipe de chercheurs qui a réalisé les interpréteurs Prolog III : Jean-Marc Boï et Frédéric Benhamou pour la partie algèbre de Boole, Pascal Bouvier pour le superviseur, Michel Henrion pour la partie numérique, Touraïvane pour le noyau de l'interpréteur et pour son travail sur la multiplication et la concaténation approchée. Je remercie également Jacques Cohen de l'Université de Brandeis dont le vif intérêt m'a poussé à écrire cet article et Franz Guenthner de l'Université de Tübingen qui m'a considérablement aidé dans la rédaction de la version anglaise de cet article. Enfin je remercie Rüdiger Loos de l'Université de Tübingen qui attiré mon attention sur deux problèmes particulièrement intéressants : la suite périodique et le remplissage d'un rectangle par des carrés.

REFERENCES

1. BALINSKI M. L. et R. E. GOMORY, A Mutual Primal-Dual Simplex Method, *Recent Advances in Mathematical Programming*, McGraw-Hill, pp. 17-26, 1963.

2. BENHAMOU F. et J-M. BOÏ, *Le traitement des contraintes booléennes dans Prolog III*, Thèses de Doctorat, GIA, Faculté des Sciences de Luminy, Université Aix-Marseille II, Novembre 1988.

3. BLAND R. G., New finite pivoting for the simplex method, *Mathematics of Operations Research*, Vol. 2, No. 2, Mai 1977.

4. BOOLE G., *The Laws of Thought*, Dover Publication Inc., 1958.

5. BROWN M., Problème proposé dans : *The American Mathematical Monthly*, vol. 90, no. 8, pp. 569, 1983.

6. BÜTTNER W. ET H. SIMONIS, Embedding Boolean Expressions into Logic Programming, *Journal of Symbolic Computation*, 4, 1987.

7. CARROLL L., *Logique sans peine*, Hermann, 1966.

8. COLMERAUER A., Theoretical model of Prolog II, *Logic programming and its*

25. SPRAGUE R., Über die Zerlegung von Rechtecken in lauter verschiedene Quadrate, *Journal für die reine und angewandte Mathematik*, 182, 1940.