COURS DE L'INSTITUT FOURIER

JULIO RUBIO
FRANCIS SERGERAERT

4. Les aspects informatiques et algorithmiques

Cours de l'institut Fourier, tome 20 (1986), p. 87-96
<http://www.numdam.org/item?id=CIF_1986__20__87_0>

4. Les aspects informatiques et algorithmiques

12. Codage en LISP

On va donner une version simplifiée de ce que fait une "machine" LISP. Il existe un "espace de départ" constitué des objets LISP et un "pointeur de départ" qui pointe sur l'objet à "évaluer"; quand l'exécution (l'évaluation) est lancée, de nouveaux objets LISP sont créés, qui, joints à ceux de départ, constituent "l'espace d'arrivée"; il existe un "pointeur d'arrivée" qui pointe le résultat (la valeur) de l'objet évalué.

Par exemple, dans le cas très simple de l'addition de deux nombres, on peut écrire :

Les objets LISP sont :

a) des atomes : des suites de caractères sans espaces;

b) des listes : des suites d'atomes et de listes entre parenthèses.

Certains atomes ont en langage LISP un sens spécial : `nil`, `t`, les nombres, `car`, `quote`,...

Quelques atomes spéciaux sont considérés comme fonctions : s'ils sont en tête d'une liste, la machine LISP comprend que les éléments suivants de la liste sont des arguments de la fonction.

EXEMPLES D'ATOMES SPÉCIAUX LISP. —

1) Le symbole + utilisé précédemment; il représente l'addition arithmétique.

2) Le symbole SETQ permet de faire *pointer* un atome vers un certain objet; par exemple, l'évaluation de `(SETQ X 5)` aura pour résultat de faire *pointer* le symbole X vers l'entier 5.

3) Le symbole QUOTE prend comme argument un objet LISP et représente la fonction identité : la valeur de la fonction sera l'objet argument de QUOTE, sans évaluation. Par exemple (QUOTE (*a b*)) aura pour valeur (*a b*) . En général, on utilise l'abréviation `'(a b)` pour `(QUOTE (a b))`.

4) Le symbole DEFUN permet de définir de nouvelles fonctions créées par l'utilisateur.

5) Le symbole LAMBDA permet de définir des fonctions *anonymes.*

On peut maintenant donner une description plus précise de ce qu'est une machine LISP. L'utilisateur introduit (au clavier, par exemple) un objet LISP, vers lequel pointe le pointeur de départ. Sur demande de l'utilisateur, l'évaluation de l'objet introduit commence; quand elle est terminée, le pointeur d'arrivée pointe sur un objet LISP qui est la valeur de l'objet introduit. Il est très important de remarquer que, pendant l'évaluation, non seulement la valeur est créée, mais que d'autres objets LISP peuvent être construits, et d'autres travaux peuvent être exécutés; ce seront des *effets de bord.* Par exemple, l'évaluation de la liste (PRINT 'A 'B) donne comme valeur B, mais comme effets de bord, les symboles A B apparaissent à l'écran.

Les règles d'évaluation d'objets LISP sont les suivantes :

a) **atomes** :

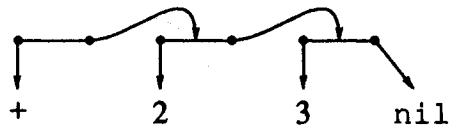
* les nombres, nil et t donnent comme valeur eux-mêmes.

* un symbole donne comme valeur l'objet vers lequel il pointe, après avoir été utilisé par l'utilisateur dans une expression utilisant la fonction SETQ; sinon la machine Lindiquera une erreur "variable non-définie".

b) **listes** :

La machine Ltrouvera en tête de liste une fonction et donnera la valeur appropriée; si la tête de liste n'est pas une fonction prédéfinie ou définie par l'utilisateur, la machine Lindiquera une erreur "fonction non définie".

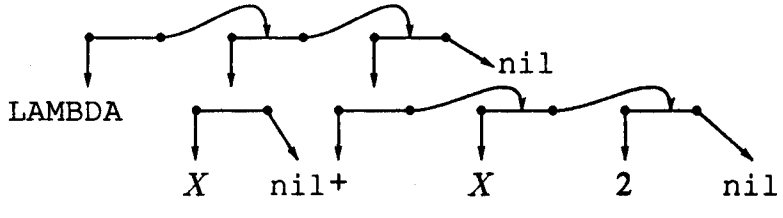
Les listes sont stockées comme des graphes au moyen de "CONS" : on peut imaginer qu'un CONS est un couple de pointeurs qui pointent chacun vers un atome ou vers un autre CONS . Le symbole nil indique la fin de la liste. Par exemple, la liste (+ 2 3) est représentée par le graphe :



Une liste plus compliquée comme

(LAMBDA (X) (+ X 2))

aurait pour représentation



(en fait, chaque atome est *unique* dans l'espace L : il n'existe qu'un X ou un nil et en général les graphes qui représentent des listes LISP *ne* sont *pas* des arbres).

Le langage LISP est commode pour travailler en Topologie Algébrique, puisqu'il permet de coder les objets complexes qui apparaissent dans cette théorie, par exemple les ensembles simpliciaux, les complexes de chaînes ou les multicomplexes.

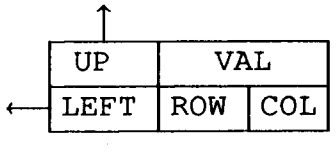
Une autre raison de choisir ce langage est qu'en homologie effective interviennent naturellement des algorithmes ayant pour données des algorithmes et donnant comme résultat un autre algorithme. Ces algorithmes ont un simple traitement en LISP comme on peut le voir dans l'exemple suivant, où est écrit un programme (une liste) LISP dont les arguments sont deux fonctions (algorithmes) à un argument et dont le résultat est la fonction (algorithme) composée des deux précédentes.

Remarque. —

```
COMPOSER
        ↓
 (LAMBDA (f g)
   (PROG (ff gg)
         (SETQ ff (gensym))
         (SETQ gg (gensym))
         (SET ff f)
         (SET gg g)
         (LIST 'LAMBDA '(X)
            (LIST ff (LIST gg 'X)))))
```

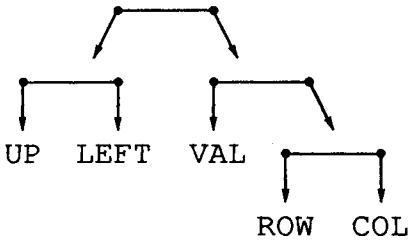
13. Matrices creuses et homologie

Les matrices utiles en Topologie Algébrique sont des matrices à coefficients entiers fréquemment *creuses* (sparses, en anglais). Quand on travaille sur machine avec ce type de matrices il est préférable, pour diminuer la complexité en espace et en temps des programmes, de ne représenter en machine que les termes non nuls. Chacun de ces termes aura cette structure :



où VAL est la valeur de l'élément (on suppose VAL$\neq 0$),
ROW est l'indice de ligne de l'élément,
COL est l'indice de colonne de l'élément,
UP pointe vers l'élément (non nul) supérieur et le plus proche sur la colonne COL et
LEFT pointe vers l'élément (non nul) à gauche et le plus proche sur la ligne ROW.

Si le langage de programmation utilisé est LISP, il faudra regarder cette structure, c'est-à-dire chaque élément non nul de la matrice, comme un CONS :

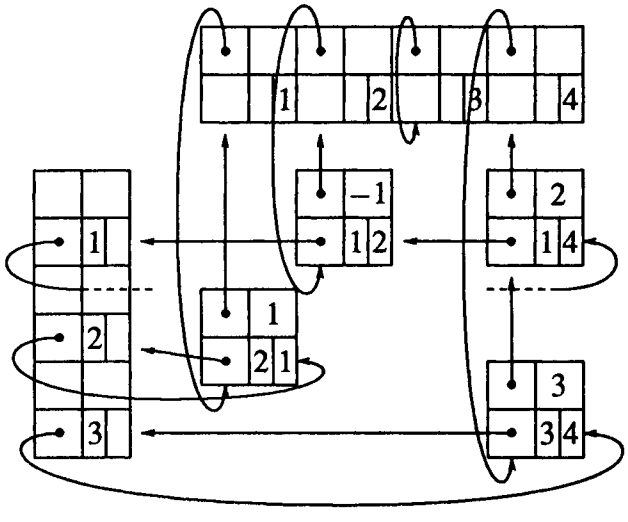


Il est par ailleurs nécessaire de créer une "ligne zéro" et une "colonne zéro", constituées de "pseudo-termes", dont la valeur est nulle, pour résoudre les problèmes de début et fin de liste.

Ainsi, une matrice comme

$$\begin{bmatrix} 0 & -1 & 0 & 2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

serait représentée par :



Un cas où les matrices creuses apparaissent d'une façon naturelle est celui où on essaie de calculer les groupes d'homologie en Algèbre Homologique ou Topologie Algébrique. Dans le cas plus simple on a un couple f, g de morphismes de $\mathbf{Z}$-modules libres :

$$\mathbf{Z}^a \xleftarrow{f} \mathbf{Z}^b \xleftarrow{g} \mathbf{Z}^c$$

tel que $f \circ g = 0$ et on définit leur *groupe d'homologie* $H(f,g) = \operatorname{Ker} f / \operatorname{Im} g$. Au moyen de changements de base dans $\mathbf{Z}^a$ et $\mathbf{Z}^b$ on peut trouver une matrice coordonnée de f du type $\operatorname{diag}[d_1, \ldots, d_p, 0, \ldots, 0]$, où $d_i > 0$ et d_i divise d_{i+1} ; d'une façon analogue, on peut trouver pour g une matrice $\operatorname{diag}[e_1, \ldots, e_q, 0, \ldots, 0]$, où $e_i > 0$ et e_i divise e_{i+1} . Alors; $H(f,g) \cong \mathbf{Z}^{b-p-q} \oplus \mathbf{Z}_{e_1} \oplus \cdots \oplus \mathbf{Z}_{e_q}$.

On remarque donc que l'étape essentielle pour le calcul d'un groupe d'homologie consiste à trouver la diagonalisation d'un morphisme de $\mathbf{Z}$-modules. Un théorème connu

(voir [V]) dit que toute **Z**-matrice peut être réduite à une matrice diagonale du type indiqué au moyen d'une suite finie de modifications parmi les suivantes :

1) changer le signe d'une ligne (ou colonne)

2) échanger deux lignes (ou deux colonnes)

3) ajouter à une ligne (ou colonne) un multiple d'une autre ligne (ou colonne, respectivement).

En plus, la démonstration du théorème donne un algorithme pour trouver la matrice diagonale cherchée (algorithme du pivot).

Dans les cas usuels en Topologie Algébrique, les matrices de départ sont creuses (c'est le cas des matrices d'incidence des complexes simpliciaux) et il faut donc réinterpréter les étapes 1), 2), 3) dans le cas des matrices creuses.

En développant ces idées, le deuxième auteur a construit une collection de programmes en langage LISP pour le traitement des matrices creuses et le calcul de l'homologie des ensembles simpliciaux. Ces programmes sont raisonnablement efficaces. Par exemple, le calcul du groupe d'homologie $H_4\big(K(\mathbf{Z}_2,2)\big)$, qui travaille sur deux matrices de 8×64 et 64×1024 éléments (coefficients de remplissage 33% et le 7% respectivement) est rapide : $H_4\big(K(\mathbf{Z}_2,2)\big) = \mathbf{Z}_4$ est calculé avec un générateur explicite, en 8 secondes CPU.

14. Métamatrices et homologie effective

Malgré le progrès représenté par les techniques de matrices creuses exposées dans le paragraphe précédent, les matrices usuelles de la Topologie Algébrique dépassent largement les possibilités mémoire des machines actuelles. Pourtant, ceci n'empêche pas de travailler avec de telles matrices; c'est ce qu'il s'agit d'expliquer très succintement maintenant.

On dira qu'un objet mathématique (matrice, nombre, ensemble, **Z**-module ,...) est *représentable* s'il peut être représenté sur machine d'une façon habituelle.

Cette "définition" est évidemment fort imprécise. Par exemple, on dira qu'une matrice n'est pas représentable si on ne peut pas la représenter comme un tableau de dimension deux ou comme un graphe selon la méthode indiquée dans le paragraphe précédent pour les matrices creuses. Un cas d'ensemble non représentable est celui de l'ensemble des nombres réels représentables sur un ordinateur donné.

Pourtant, même les objets mathématiques non représentables peuvent être utilisés sur machine à condition qu'ils soient algo–(ou méta–) objet.

DÉFINITIONS. — *Un* **Z***-module libre A muni d'une base* $\{a_i\}_{i\in I}$ *indexée par un ensemble I est un algo–module si on dispose d'un algorithme qui pour chaque* $i \in I$

sait calculer a_i .

Si A et B sont deux algo–modules et si $f : A \to B$ est un morphisme, on dira que f est une *algo–morphisme* si on dispose d'un algorithme qui, pour chaque $i \in I$, calcule $f(a_i)$, où $\{a_i\}_{i\in I}$ est la base choisie pour A .

D'une façon analogue, on peut définir quelques autres algo–objets mathématiques; par exemple, les algo–complexes de chaînes.

L'idée de matrice coordonnée d'un algo–morphisme donne la suivante :

DÉFINITION. — *Une métamatrice à m lignes et n colonnes est un algorithme permettant d'associer à tout couple d'entiers $1 \leqslant p \leqslant m$, $1 \leqslant q \leqslant n$, une valeur.*

Un exemple simple d'une métamatrice, dont la matrice associée est *non* représentable, est donné par l'expression `LISP` suivante :

```
(DEFUN terme (p q) (+ p q)).
```

La raison d'introduire de telles notions (et aussi l'homologie effective) vient du problème du traitement et du calcul sur machine de l'homologie des modèles simpliciaux des espaces d'Eilenberg–Mac Lane : ils sont tellement énormes qu'ils sont *non* représentables sur toute machine. Mais il est facile de montrer que les ensembles simpliciaux (K, π, p) , $\widetilde{K}(\pi, p)$ des exemples 5.10 (5) et (6) ont des algo–complexes de chaînes associés, autrement dit que les opérateurs de bord de leurs complexes de chaînes associés sont des algo–morphismes (ou, d'une autre façon, leurs matrices d'incidence sont des métamatrices).

Si on a un algo-complexe de chaînes non représentable (C, d) avec lequel on souhaite travailler et que l'on suppose son homologie effective (HC, d) calculée alors on peut espérer que les réductions faites sur C feront que l'ensemble de générateurs de HC_* soit déjà représentable de même que les métamatrices associées à (HC, d) . Alors on pourra appliquer à (HC, d) les méthodes de la section 13 et trouver l'homologie de (C, d) .

De plus les méthodes de l'homologie effective peuvent être utilisées pour le calcul d'autres invariants en Topologie Algébrique; notamment, les groupes d'homotopie de CW-complexes finis, avec la version effective de la suite spectrale de Serre, comme développé au le paragraphe 8.

On explique maintenant dans quelles conditions les algo-morphismes peuvent être composés (autrement dit, les métamatrices multipliées) et comment ceci peut être appliqué au cas des suites exactes courtes. Ceci, et la simplicité de l'exemple final de 12, nous donne une autre raison pour le choix du langage `LISP` pour les applications concrètes sur machine.

Soient A, B deux métamatrices $m \times n, n \times p$ respectivement. On suppose en plus qu'on a un algorithme qui à chaque indice k de colonne de B lui fait correspondre les indices j de ligne tels que $b_{jk} \neq 0$ (il faut donc supposer que l'ensemble de ces indices j est représentable); il est clair alors que, étant donné i, k, on a un algorithme pour

déterminer $(AB)_{i,k} = \sum_j a_{ik} b_{jk}$; c'est-à-dire , dans ces conditions le produit de deux métamatrices est une autre métamatrice.

On reprend maintenant les notations du paragraphe 3 et celles de la démonstration de la proposition 3.7 : (i, p, j, q) est une algo-suite exacte courte entre les algo-complexes de chaînes $(A_*, \overline{d})$, (B_*, d) , $(C_*, \overline{\overline{d}})$; on connaît l'homologie effective $(HA_*, \overline{d})$, $(HC_*, \overline{\overline{d}})$, mais (B_*, d) est *non*-représentable. L'homologie effective de (B_*, d) est (d'après la proposition 3.7) celle du multicomplexe HD_* donné par :

$$\begin{array}{ccc}
\vdots & & \vdots \\
\downarrow & & \downarrow \\
HA_n & & H\overline{C}_n \\
{\scriptstyle \overline{d}_n}\downarrow & \overline{f}_n \swarrow & \downarrow{\scriptstyle \overline{\overline{d}}_n} \\
HA_{n-1} & & HC_{n-1} \\
\downarrow & & \downarrow \\
\vdots & & \vdots
\end{array}$$

où $\overline{f}_n$ est le composé :

$$HC_n \xrightarrow{a} C_n \xrightarrow{j_n} B_n \xrightarrow{d_n} B_{n-1} \xrightarrow{q_{n-1}} A_{n-1} \xrightarrow{b} HA_{n-1} \ .$$

Alors a, j_n, d_n, q_{n-1}, b sont des métamatrices, mais si, (comme on peut l'espérer) HC_* et HA_* sont représentables (c'est-à-dire si leurs ensembles de générateurs sont représentables), alors $\overline{f}_n$ est aussi une métamatrice, représentable en tant que telle. Le complexe HD est représentable et, par réduction flèche à flèche, on peut en calculer sur machine l'homologie effective, donc l'homologie de (B_*, d) .

Références pour le Chapitre 4

[V] VEBLEN O.. — *Analysis Situs*, A.M.S., Publ. Vol. 5, part.II, 1931.