RAIRO
INFORMATIQUE THÉORIQUE

RAUL KANTOR
GIOVANNA SONTACCHI

Un interprète LISP de la programmation fonctionnelle réalisé par des combinateurs

RAIRO – Informatique théorique, tome 19, n° 1 (1985), p. 33-41.
<http://www.numdam.org/item?id=ITA_1985__19_1_33_0>

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 19, n° 1, 1985, p. 33 à 41)

UN INTERPRÈTE LISP DE LA PROGRAMMATION FONCTIONNELLE RÉALISÉ PAR DES COMBINATEURS (*)

par Raul KANTOR (1) et Giovanna SONTACCHI (1)

Communiqué par G. AUSIELLO

Résumé. — *Cet article décrit l'implémentation d'une variante, due à Böhm, de la programmation fonctionnelle de Backus et qui utilise un ensemble de combinateurs. A cet effet on développe un système de réécriture pour aboutir à un programme écrit en LISP.*

Abstract. — *The paper describes the implementation of a Böhm variant of Backus Functional Programming through a set of combinators. For this aim a rewriting system is throughout developed into a computer program written in LISP.*

0. INTRODUCTION

Cette communication se situe dans le cadre de la Programmation Fonctionnelle (FP), et plus proprement dans la partie qui lie cette théorie avec la Logique Combinatoire (LC).

La relation qui existe entre la proposition révolutionnaire de Backus [1] et la LC (ou, pour mieux dire, la « relecture » de la FP au moyen de la LC), relevée et en outre approfondie par Böhm [2, 3], nous permet d'entrevoir des possibilités et des développements d'intérêts remarquables.

En effet, d'une part la rigueur qui caractérise la LC, pour sa même nature, permet de poser sur des bases plus solides du point de vue formel les clairvoyantes intuitions de Backus; d'autre part, le caractère extrêmement essentiel de la LC, qui s'exprime au moyen d'une économie remarquable de ressources, est très exploité dans ce cadre.

On réussit de cette façon à démontrer la possibilité d'exprimer les opérateurs proposés en [1] (et tous ceux qui, bien plus nombreux, paraissent dans les

(*) Reçu janvier 1984, révisé juin 1984.
Recherche partiellement financée par le M.P.I. et par le C.N.R. à l'intérieur du Projet finalisé pour l'Informatique.
(1) Istituto di Matematica, Università "La Sapienza", Roma.

articles qui s'y rapportent) au moyen d'un nombre assez réduit d'opérateurs fondamentaux.

Cela nous permet, d'un point de vue qu'on pourrait définir d'intérêt « théorique », de relever, en utilisant ce qui paraît comme une « factorisation » des expressions, les rapports entre deux opérateurs apparemment sans relations, avec la possibilité d'effectuer des simplifications; celles-ci sont le but principal de l'approche algébrique que Backus donne à son œuvre; en même temps, d'un point de vue qu'on pourrait dire, par opposition, d'intérêt « technique », le nombre réduit d'opérateurs impliqués, laisse entrevoir la possibilité d'une réalisation physique de la « machine » de Backus.

Selon cette réflexion nous avons pensé qu'une interprétation du langage combinatoire au moyen d'algorithmes qui sont réalisables avec des machines de von Neúmann peut garder un certain intérêt, en parvenant à un approche qui se différencie des simulations de la machine de Backus exécutées sur des ordinateurs réels [4] par le fait de travailler avec les éléments essentiels proposés par Böhm (un tel choix approche, dans ce sens-ci, notre communication aux propositions du SKIM [5]).

Enfin quelques remarques sur la structure de la communication : dans le paragraphe 1, en utilisant les idées principales des articles de Böhm, on pourvoit à formaliser ultérieurement quelques notions en vue d'une mise en œuvre effective de l'algorithme de réécriture; dans le deuxième paragraphe on procède à la description des procédures qui composent tel algorithme et à la traduction du même dans un programme concret, écrit en langage LISP, tandis que dans la conclusion on se propose de décrire quelques possibles développements des résultats obtenus.

1. FORMALISATION DU PROBLÈME

Le développement de la LC a conduit à la définition de différents systèmes et il a permis plusieurs approches qui, tout en étant au fond équivalentes, se caractérisent par la particulière attention consacrée, de temps en temps, à un ou à plusieurs aspects (naturel des définitions, non-redondance, élégance du système) de la construction formelle.

Le modèle que nous avons choisi est essentiellement celui qui est proposé dans [6], qui se signale par sa concision et son expressivité, et qui semble particulièrement fonctionnel, comme il est démontré dans [2, 3], au développement en termes de LC de la FP; de plus, de notre point de vue, l'efficacité qu'un tel modèle révèle en termes d'automatisation des règles de réécriture des expressions, prend un relief particulier.

Nous commençons donc en présentant brièvement un tel modèle.

On a, en premier lieu :

1.1. L'alphabet $\mathscr{A}$ est ainsi défini :

$$\mathscr{A} = \mathbb{N} \cup \mathscr{O} \cup \mathscr{C},$$

où $\mathbb{N}$ est l'ensemble des nombres naturels, l'ensemble $\mathscr{O}$ des opérateurs est le suivant : $\{ +, -, *, :, Z, unit, hd, tl, cat, / \}$ et l'ensemble $\mathscr{C}$ des combinateurs est le suivant : $\{ I, O, C_*, K, B, A, D \}$.

1.2. L'ensemble $\mathscr{T}$ des termes est défini de la façon suivante :

(1) $a \in \mathscr{A} \Rightarrow a \in \mathscr{T}$;

(2) $\forall n \geqq 1,\ t_1, \ldots, t_n \in \mathscr{T} \Rightarrow (t_1 \ldots t_n) \in \mathscr{T}$;

(3) $\forall n \geqq 0,\ t_1, \ldots, t_n \in \mathscr{T} \Rightarrow (\# t_1 \ldots t_n) \in \mathscr{T}$.

Nous appellerons liste chaque terme de la forme $(\# t_1 \ldots t_n)$.

REMARQUE : Des parenthèses superflues peuvent paraître dans les termes [par exemple $((\# t_1 t_2 \ldots t_n))$ est équivalent à $(\# t_1 t_2 \ldots t_n)$, ((5)) est équivalent à 5, etc.]; de plus, car vaut l'associativité à gauche de l'opération sous-entendue de l'application, nous pouvons considérer superflues les parenthèses qui explicitent les relatives précédences [par exemple $(((t_1 t_2) t_3) t_4)$ est équivalent à $(t_1 t_2 t_3 t_4)$, etc.].

L'ensemble $\mathscr{R}$ des règles qui déterminent le comportement des combinateurs et des opérateurs peut s'exprimer de la façon suivante :

$\mathscr{R}$:

(r 1) $I t = t$;

(r 2) $O t_1 t_2 = t_2$;

(r 3) $C_* t_1 t_2 = t_2 t_1$;

(r 4) $K t_1 t_2 = t_1$;

(r 5) $B t_1 t_2 t_3 = t_1 (t_2 t_3)$;

(r 6) $A t_1 t_2 t_3 = B (t_1 t_3) (t_2 t_3)$;

(r 7) $D t_1 t_2 t_3 = t_3 t_1 t_2$;

(r 8) $+ n_1 n_2 = n_3$ où $n_3 = n_1 + n_2$;

(r 9) $- n_1 n_2 = n_3$ où $n_3 = n_1 - n_2$ si $n_1 > n_2$, $n_3 = 0$ autrement;

(r 10) $* n_1 n_2 = n_3$ où $n_3 = n_1 * n_2$;

(r 11) $: n_1 n_2 = n_3$ où $n_3 = 0$ si $n_2 = 0$, $n_3 = [n_1 / n_2]$ autrement;

(r 12) $Z n t_1 t_2 =$ t_2 si $n = 0$, / $t_1 (Z m t_1 t_2)$ où $m = n - 1$ autrement; \

$$(r\,13)\quad unit\ t=(\#\,t);$$

$$(r\,14)\quad hd\,l=\begin{cases}(\#) \text{ si } l=(\#),\\ t_1 \text{ si } l=(\#\,t_1\ldots t_n);\end{cases}$$

$$(r\,15)\quad tl\ l=\begin{cases}(\#) \text{ si } l=(\#),\\ (\#\,t_2\ldots t_n) \text{ si } l=(\#\,t_1\,t_2\ldots t_n);\end{cases}$$

$$(r\,16)\quad cat\,l_1\,l_2=(\#\,t_1\ldots t_n\,t'_1\ldots t'_m) \text{ si } l_1=(\#\,t_1\ldots t_n) \text{ et } l_2=(\#\,t'_1\ldots t'_m);$$

$$(r\,17)\quad /l\,t_1\,t_2=\begin{cases}t_2 \text{ si } l=(\#),\\ t_1\,(hd\,l)\,(/(tl\,l)\,t_1\,t_2) \text{ autrement}\end{cases}$$

où t, $t_i\in\mathscr{T}$, n, m, $n_i\in\mathbb{N}$, l, l_i sont des listes.

Nous pouvons donner maintenant la définition de l'ensemble $\mathscr{T}_{\mathscr{A}}$ des termes admissibles :

1.3. L'ensemble $\mathscr{T}_{\mathscr{A}}$ des termes admissibles est défini de la façon suivante :

(1) $t\in\mathbb{N}\Rightarrow t\in\mathscr{T}_{\mathscr{A}}$;

(2) $t_1, \ldots, t_n\in\mathscr{T}_{\mathscr{A}}\Rightarrow(\#\,t_1\ldots t_n)\in\mathscr{T}_{\mathscr{A}}$;

(3) $t\in\mathscr{T}$ et $\mathscr{R}(t)\in\mathscr{T}_{\mathscr{A}}\Rightarrow t\in\mathscr{T}_{\mathscr{A}}$, où $\mathscr{R}(t)$ est le résultat de l'application réitérée des règles de réduction $\mathscr{R}$ à t.

Nous présentons un algorithme de réécriture (RIDD) qui distingue les termes admissibles des termes non admissibles : dans le premier cas il donne en output le terme réduit correspondant, dans le deuxième cas un message d'erreur.

2. DESCRIPTION DE L'ALGORITHME

Dans ce paragraphe nous allons décrire l'algorithme RIDD de réécriture (réduction). La stratégie de réduction adoptée par nous est essentiellement du type outer-most left-most, avec recours, s'il y a lieu, à techniques du type left-most inner-most.

Cette stratégie prévoit l'application des opérations suivantes sur le terme à réduire :

(1) élimination des parenthèses superflues (procédures SUBA et SUBB);

(2) identification du premier sous-terme à gauche et application, s'il est possible, de la règle de réduction relative (procédures S 1, S 2 et OP);

(3) application récursive de (1) et (2) au terme réduit.

A propos du pas (2) de l'algorithme, on remarque que les conditions suivantes peuvent se présenter :

(*a*) le sous-terme plus à gauche est un nombre et coïncide avec le terme même : alors le réduit est le nombre même;

(*b*) le sous-terme plus à gauche est un combinateur, et alors on procède à l'identification du rédèxe;

(*c*) le sous-terme plus à gauche est un opérateur, et alors on procède à l'identification des opérands et en les réduisant par RIDD;

(*d*) le sous-terme plus à gauche est une liste et coïncide avec le terme même, et alors on procède à l'application de RIDD aux éléments de la liste;

(*e*) aucune des conditions (*a*), (*b*), (*c*), (*d*) n'est vérifiée, et alors on a un message d'erreur.

Nous voulons souligner enfin le rôle de la procédure OUT (contrôle final sur la nature du terme réduit) dans l'algorithme RIDD.

Nous allons donner maintenant la liste du programme, en avisant que, pour des raisons typographiques, dans la liste l'on a écrit *C* à la place de C_*.

```
@$@$@$*SONTAC(1).POP
      1       @lisp
      2       (csetq ridd (lambda (l)
      3             (cond ((atom l) (out l))
      4                   (t (out (subb (suba (rid l))))))))
      5       (csetq rid (lambda (l)
      6            (cond ((atom l)l)
      7                  (t (s1 (subb (suba l)))))))
      8       (csetq s1 (lambda (l)
      9            (cond ((and (equal 1 (length l))(atom (car l)))l)
     10                  ((not (atom (car l))) 'error)
     11                  ((equal '# (car l))(cons '# (s2 (cdr l))))
     12                  ((equal 'i (car l))(rid (opi l)))
     13                  ((equal 'unit (car l))(rid (opunit l)))
     14                  ((equal 'tl (car l))(rid (optl l)))
     15                  ((equal 'hd (car l))(rid (ophd l)))
     16                  ((equal 2 (length l)) (suba (cons (car l)(s2 (cdr l)))))
     17                  ((equal 'k (car l))(rid (opk l)))
     18                  ((equal 'o (car l))(rid (opo l)))
     19                  ((equal 'c (car l))(rid (opc l)))
     20                  ((equal '+ (car l))(rid (op+ l)))
     21                  ((equal '- (car l))(rid (op- l)))
     22                  ((equal '* (car l))(rid (op* l)))
     23                  ((equal ': (car l))(rid (op: l)))
     24                  ((equal 'cat (car l))(rid (opcat l)))
     25                  ((equal 3 (length l)) (suba (cons (car l)(s2 (cdr l)))))
     26                  ((equal 'a (car l))(rid (opa l)))
     27                  ((equal 'b (car l))(rid (opb l)))
     28                  ((equal 'd (car l))(rid (opd l)))
     29                  ((equal 'z (car l))(rid (opz l)))
     30                  ((equal '/ (car l))(rid (op/ l)))
     31                  (t (suba (cons (car l)(s2 (cdr l)))))))))
```

```
(csetq s2 (lambda (l)
     (cond ((null l) l)
           (t(append(list(rid(car l)))(s2(cdr l)))))))
(csetq l#n (lambda (l)
     (cond ((not (equal '# (car l))) f)
           ((and (equal '# (car l))(null (cdr l)))t)
           ((ln (cdr l))t)
           (t f))))
(csetq ln (lambda (l)
     (cond ((not (numberp (car l))) f)
           ((null (cdr l)) t)
           ((ln (cdr l))t)
           (t f))))
(csetq ll#n (lambda (l)
     (cond ((atom l)f)
           ((not (equal '# (car l)))f)
           ((l#n l)t)
           ((lln (cdr l))t)
           (t f))))
(csetq lln (lambda (l)
     (cond ((not (or (numberp (car l))(ll#n (car l))))f)
           ((null (cdr l))t)
           ((lln (cdr l))t)
           (t f))))
(csetq out (lambda (l)
     (cond ((numberp l)l)
           ((atom l) 'error)
           ((and (equal 1 (length l))(numberp (car l)))(car l))
           ((ll#n l)l)
           (t 'error))))
(csetq suba (lambda (l)
     (cond ((null l) l)
           ((atom (car l))(cons (car l)(suba (cdr l))))
           ((and (equal 1 (length (car l)))(equal '(#)(car l)))
                (cons (car l)(suba (cdr l))))
           ((equal 1(length (car l)))(append (suba (car l))
               (suba (cdr l))))
           (t (append (list (suba (car l)))(suba (cdr l)))))))
(csetq subb (lambda (l)
     (cond ((null l)l)
           ((atom (car l))l)
           ((equal 1 (length l))(subb (car l)))
           ((equal '# (car (subb (car l))))(append (list (subb
               (car l)))(cdr l)))
           (t (append (subb (car l))(cdr l))))))
(csetq opi (lambda (l)
     (cdr l)))
(csetq opo (lambda (l)
           (cddr l)))
(csetq opc (lambda (l)
     (cons (caddr l)(cons (cadr l)(cddddr l)))))
(csetq opk (lambda (l)
     (cons (cadr l)(cddddr l))))
(csetq op+ (lambda (l)
     (cond ((and (opnum (rid (list (cadr l))))(opnum (rid (list
           (caddr l)))))(cons (plus (car (rid (list
           (cadr l))))(car (rid (list (caddr l)))))(cddddr l)))
           (t 'error))))
(csetq opa (lambda (l)
     (append (list 'b (list (cadr l)(cadddr l))(list (caddr l)
           (cadddr l)))(cddddr l))))
```

```
93      (csetq opb (lambda (l)
94           (append (list (cadr l)(list (caddr l)(cadddr l)))(cddddr l))))
95      (csetq opd (lambda (l)
96           (append (list (cadddr l)(cadr l)(caddr l))(cddddr l))))
97      (csetq op* (lambda (l)
98           (cond ((and (opnum (rid (list (cadr l))))(opnum (rid (list
99                       (caddr l))))(cons (times (car (rid (list
100                      (cadr l))))(car (rid (list (caddr l))))(cdddr l)))
101                (t 'error))))
102     (csetq op: (lambda (l)
103          (cond ((and (opnum (rid (list (cadr l))))(opnum (rid (list
104                      (caddr l))))(cons (quotient (car (rid (list
105                      (cadr l))))(car (rid (list (caddr l))))(cdddr l)))
106                (t 'error))))
107     (csetq op- (lambda (l)
108          (cond ((and (opnum (rid (list (cadr l))))( opnum (rid (list
109                      (caddr l))))(cons (dif0 (car (rid (list
110                      (cadr l))))(car (rid (list (caddr l))))(cdddr l)))
111                (t 'error))))
112     (csetq opz (lambda (l)
113          (cond ((and (opnum (rid (list (cadr l))))(equal 0 (car (rid
114                   (list (cadr l))))) (cdddr l))
115                ((opnum (rid (list (cadr l))))(append (list (caddr l)(list 'z
116                   (sub1 (car (rid (list (cadr l))))(caddr l)(cadddr l)))
117                   (cddddr l)))
118                (t 'error))))
119     (csetq oplis (lambda (l)
120          (cond ((atom l) nil)
121                ((null l) nil)
122                ((equal '# (car  l)) t)
123                ((atom (rid l)) nil)
124                ((equal '# (car (rid l))) t)
125                (t nil))))
126     (csetq opunit (lambda (l)
127          (cons (list '# (cadr l))(cddr l))))
128     (csetq ophd (lambda (l)
129          (cond ((null (oplis (cadr l))) 'error)
130                ((equal '(#)(rid (cadr l))) '(#))
131                (t (cons (cadr (rid (cadr l)))(cddr l))))))
132     (csetq optl (lambda (l)
133          (cond ((null (oplis (cadr l))) 'error)
134                ((equal '(#)(rid (cadr l)))'(#))
135                (t(cons (cons '# (cddr (rid (cadr l)))) (cddr l))))))
136     (csetq opcat (lambda (l)
137          (cond ((null (oplis (cadr l))) 'error)
138                ((null (oplis (caddr l))) 'error)
139                (t(cons(append(rid(cadr l))(cdr(rid(caddr l))))(cdddr l))))))
140     (csetq op/ (lambda (l)
141          (cond ((null (oplis (cadr l))) 'error)
142                ((equal 1 (length (rid (cadr l)))) (cddddr l))
143                (t(append (list (caddr l)(rid (list 'hd (cadr l)))(list '/
144                   (rid(list 'tl(cadr l)))(caddr l)(cadddr l)))(cddddr l))))))
145     (csetq opnum (lambda (l)
146          (cond ((and (equal 1 (length (rid (list l))))
147                   (numberp (car (rid (list l)))))t)
148                (t nil))))
149     (csetq dif0 (lambda (x y)
150          (cond ((greaterp x y)(difference x y))
151                (t '0))))
```

Nous donnons maintenant quelques exemples de réduction opérées par le programme.

```
(ridd'(Z 4 UNIT 7))
VALUE: (#(#(#(#7))))
EVAL:
(ridd'(Z 5(+1) 0))
VALUE: 5
EVAL:
(ridd'(/(#6 (#1 2) 3 7) (K(+1)) 0))
VALUE: 4
EVAL:
(ridd'(/(#1 2 3 4 5) (B(A(A C(K B)) (K C) CAT) UNIT) (#)))
VALUE: (#5 4 3 2 1)
EVAL:
(ridd'(/(#1 2 3 4 5) (B(B CAT UNIT) (*2)) (#)))
VALUE: (#2 4 6 8 10)
EVAL:
(ridd'(/(#(+10) (*2) (-100)  (:80)) (B(B(B CAT UNIT)) C 10) (#)))
VALUE: (#20 20 90 8).
```

Dans les deux premiers exemples on montre l'usage de l'opérateur Z comme itérateur.

Dans les exemples suivants on emploie l'opérateur / pour obtenir quatre buts différents :

(1) la longueur d'une liste non linéaire;

(2) le renversement d'une liste;

(3) l' « apply-to-all » du doublement à une liste numérique;

(4) l'application de la « construction » d'une liste de fonctions arithmétiques à l'entier 10.

CONCLUSION

Le programme aurait pu être amélioré du point de vue des temps de calcul, mais le but de notre étude est différent.

En effet il semble inutile d'aller dans cette direction, car ce qu'on a fait peut être vu comme une partie d'un interprète du langage de Backus pour n'importe quelle machine de type von Neumann; il s'agit donc de la simulation d'une machine à présent idéale, avec des caractéristiques complètement nouvelles, sur une machine réelle non douée de telles propriétés.

Nous voudrions en outre souligner que la réduction combinatoire définie dans notre étude correspond à la réduction dite « faible » en logique combinatoire : en effet, par exemple, les expressions BXI et BIX sont irréductibles même s'ils se réduisent fortement à X. Clairement une amélioration dans ce sens est possible si l'on ajoute l'algorithme de la réduction forte [8].

REMERCIEMENTS

Nous remercions en particulier le professeur Böhm et le docteur Berarducci, qui ont discuté avec nous de façon constante et critique les développements successifs de notre travail, et le professeur Robinet pour les conseils précieux.

BIBLIOGRAPHIE

1. J. Backus, *Can Programming be Liberate from von Neumann Style? A Functional Style and its Algebra of Programs*, C.A.C.M. vol. 21, n° 8, 1978, p. 577-602.
2. C. Böhm, *The Purpose of Unit-List in Functional Programming*, Conference on Information Sciences, Oberwolfach, 1982.
3. C. Böhm, *Combinatory Foundation of Functional Programming*, A.C.M. Symposium on L.I.S.P. and Functional Programming, vol. 18, 15 août 1982, p. 29-36.
4. A. Chiarini, *On FP Languages Combining Forms*, SIGPLAN, vol. 15, n° 9, 1980, p. 25-27.
5. T. J. W. Clark, P. J. S. Gladstone, C. D. McLean et A. C. Norman, *SKIM. The S,K,I Reduction Machine*, Conf. Rec. of the 1980 L.I.S.P. Conf., Stanford, CA, août 1980, p. 128-135.
6. C. Böhm, *Un modèle arithmétique des termes de la logique combinatoire, Lambda calcul et sémantique formelle des langages de programmation*, Actes de la Sixième École de Printemps d'Informatique Théorique, La Châtre, B. Robinet, éd. L.I.T.P. and E.N.S.T.A., Paris, 1979, p. 97-108.
7. C. Böhm, *An abstract Approach to (Hereditary) Finite Sequences of Combinators*, H. B. Curry, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980, p. 231-242, J. P. Seldin et J. R. Hindley, éd., Academic Press, London.
8. H. B. Curry, R. Feys et W. Craig, *Combinatory Logic*, vol. 1, 1958, North Holland, Amsterdam.