RAIRO. INFORMATIQUE THÉORIQUE

G. BERRY

# Calculs ascendants du programme d'Ackermann : analyse du programme de J. Arsac

*RAIRO. Informatique théorique*, tome 11, n° 2 (1977), p. 113-126

<http://www.numdam.org/item?id=ITA_1977__11_2_113_0>

R.A.I.R.O. Informatique théorique/Theoretical Computer Science
(vol. 11, n° 2, 1977, p. 113 à 126).

# CALCULS ASCENDANTS DU PROGRAMME D'ACKERMANN : ANALYSE DU PROGRAMME DE J. ARSAC (1)

par G. BERRY (2)

Communiqué par J.-F. PERROT

Résumé. – *En employant la technique d'analyse des calculs ascendants, on caractérise la stratégie de la procédure proposée par J. Arsac pour calculer la fonction d'Ackermann, et on donne une estimation de son temps de calcul.*

## INTRODUCTION

La fonction d'Ackermann a été introduite en théorie de la récursivité comme exemple de fonction récursive totale mais non récursive primitive, c'est-à-dire non définissable par un nombre fini de récurrences simples. On l'écrit d'habitude à l'aide de la récurrence double suivante :

$$Ack(m, n) = si\ m = 0\ alors\ n + 1\ sinon\ si\ n = 0\ alors\ Ack(m - 1, 1)\ sinon\ Ack(m - 1, Ack(m, n - 1))$$

On montre que la fonction d'Ackermann croît plus vite que toute fonction récursive primitive, et que le temps de calcul $t(P, m, n)$ de tout programme $P$ la calculant possède la même propriété (voir [15, 19]) : le calcul de $Ack(m, n)$ est impossible en pratique pour $m \geqslant 6$. Plusieurs auteurs se sont cependant intéressés au *programme récursif* qui définit cette fonction : c'est un des exemples les plus simples d'appels récursifs imbriqués donnant lieu à des calculs complexes.

Pour montrer l'inefficacité intrinsèque des calculs récursifs, H. G. Rice [14] propose un programme itératif de calcul de la fonction d'Ackermann qui

(1) Reçu en février 1977.
(2) École Nationale Supérieure des Mines de Paris, Sophia Antipolis, Valbonne et Iria-Laboria, Rocquencourt.

utilise deux tableaux de taille $m$ au lieu d'une pile croissant comme $Ack(m, n)$; son programme n'effectue par ailleurs aucun recalcul de valeurs intermédiaires, alors que l'appel par valeur recalcule $Ack(0, 1)$ au moins $Ack(m, n)$ fois. Le principe de l'algorithme de Rice est d'« inverser » la définition récursive, et de calculer de manière ascendante en partant des valeurs calculables directement ($m = 0$) vers le résultat cherché : c'est ce qu'on fait d'ordinaire pour calculer les nombres de Fibonacci, ou les coefficients du binôme par le triangle de Pascal. Nous avons présenté en [3, 4] une théorie générale des calculs ascendants des programmes récursifs, qui nous permet d'étudier formellement leur complexité en temps et en mémoire. Nous y utilisons l'algorithme de Rice comme exemple, et montrons dans le modèle formel son optimalité en temps et en mémoire au sein des algorithmes ascendants associés au *programme* d'Ackermann donné ci-dessus (et non bien sûr au sein des algorithmes de calcul de la *fonction* d'Ackermann, qui peut être définie par d'autres programmes).

Partant d'un tout autre point de vue, J. Arsac [1] obtient par des transformations de programme une procédure itérative calculant la fonction d'Ackermann à l'aide d'un seul tableau de taille $m$. On ne connaît a priori ni la stratégie, ni la complexité en temps du programme obtenu. Le but de cet article est d'étudier et de résoudre ces deux problèmes. Pour cela, nous montrons que le programme d'Arsac code un algorithme ascendant au sens précédent, et nous évaluons le calcul ascendant qu'il engendre. Cette démarche possède deux types d'avantages : elle permet de s'affranchir des phénomènes de codage liés à la programmation et de remplacer la notion informelle de « stratégie » par la notion formelle de calcul ascendant qui la représente dans le modèle ; elle permet d'effectuer simplement des évaluations qui s'appliquent généralement aux programmes à des facteurs linéaires près.

Les codages réalisés par les programmes d'Arsac et de Rice sont simples : le calcul ascendant que nous associons au programme d'Arsac est une optimisation en mémoire du calcul ascendant canoniquement associé à l'appel par valeur, qui conserve les mêmes recalculs et n'est donc pas optimal en temps ; celui que nous associons au programme de Rice est une optimisation en temps et mémoire de ce même calcul de l'appel par valeur. En raison de propriétés particulières du programme d'Ackermann, l'optimisation supplémentaire en temps réalisée par le calcul de Rice n'est que linéaire ; la même optimisation aurait été exponentielle dans le cas des nombres de Fibonacci.

Le fait que le programme d'Arsac utilise moins de mémoire que celui de Rice peut être vu comme une manifestation indirecte du phénomène d'échange espace-temps [7, 9] : recalculer des valeurs intermédiaires permet d'utiliser moins de mémoire, autrement dit un calcul ascendant optimal en temps ne peut pas en général être optimal en mémoire. Ce phénomène ne se manifeste pas ici dans le modèle : les calculs d'Arsac et Rice utilisent tous deux

$m$ mémoires théoriques pour calculer $Ack(m, n)$, et le calcul de Rice est optimal en mémoire et en temps. Mais une mémoire théorique contient trois entiers, le programme de Rice doit en utiliser deux et celui d'Arsac n'a besoin d'en utiliser qu'un.

Notons que la présence des phénomènes de codage rend impossible une définition satisfaisante de l'optimalité en mémoire des programmes : dès qu'une mémoire peut contenir un entier, on peut lui faire contenir une suite d'entiers de longueur arbitraire, et calculer ainsi toute fonction récursive avec les fonctions successeur, prédécesseur, test à zéro et seulement trois mémoires (dans ce cas, nos évaluations donnent tout de même une idée de la taille des entiers contenus dans les mémoires). En revanche, nos mémoires théoriques ne sont manipulables que par une seule opération, directement dérivée du programme récursif.

Afin que le lecteur n'ait pas à se référer constamment à [3, 4], une première partie présente rapidement la notion de structure de récursion associée à un programme, la définition des calculs ascendants et la construction du calcul ascendant canoniquement associé à l'appel par valeur. La deuxième partie présente l'analyse des transformations et du programme de J. Arsac et la comparaison des algorithmes de Rice et d'Arsac. Nous développons dans la conclusion quelques réflexions sur les rapports entre transformations de programmes et analyse d'algorithmes.

## I. CALCULS ASCENDANTS DES PROGRAMMES RÉCURSIFS

### I.1. Calculs par réécriture et calculs ascendants

Contrairement à un programme itératif, un programme récursif n'est pas la spécification pas à pas d'un calcul. Deux types principaux d'algorithmes de calcul peuvent être employés pour un programme récursif donné.

*Les calculs par réécriture* [5, 11, 17]. On considère un programme comme un système de règles de réécriture qu'on peut exploiter à l'aide de diverses règles de calcul : appel par nom, appel par valeur, substitution complète, etc...

*Les calculs ascendants :* Un programme est considéré comme définissant une *structure de récursion* à laquelle plusieurs algorithmes d'exploration peuvent s'appliquer. Un exemple typique de structure de récursion est le triangle de Pascal (*fig.* 1), associé au programme récursif de calcul des coefficients du binôme :

$$c(n, p) = \textit{si } p = 0 \textit{ ou } p = n \textit{ alors } 1 \text{ sinon } c(n-1, p) + c(n-1, p-1)$$

De façon générale, une structure de récursion est définie dans le graphe de la fonction calculée (ici l'ensemble des triplets $(n, p, c(n, p))$ avec $n \geqslant 0$ et $0 \leqslant p \leqslant n$), et caractérisée par la relation « $x$ est une valeur intermédiaire

immédiate du calcul de $y$ », représentée par les flèches de la figure 1. La fermeture réflexive et transitive de cette relation est la relation « $x$ est une valeur intermédiaire du calcul de $y$ »; sous certaines conditions portant sur les fonctions de base et ici satisfaites, cette relation est un ordre partiel bien fondé tel que chaque point domine un nombre fini de points. On peut alors voir la structure de récursion comme un graphe infini sans circuit, dont l'ensemble des points sans arc entrant est donné par les conditions d'arrêt du programme; dans l'exemple, c'est l'ensemble $\{ (n, n, 1), (n, 0, 1) \mid n \geqslant 0 \}$.

Si $x$ est un point du graphe, nous appellerons *producteur minimal* de $x$, en abrégé $mp(x)$, l'ensemble des prédécesseurs immédiats de $x$, est *domaine de* $x$, en abrégé $dom(x)$, l'ensemble de tous les prédécesseurs de $x$ : $dom(x)$ est l'ensemble des valeurs intermédiaires nécessaires et suffisantes pour le calcul de $x$.

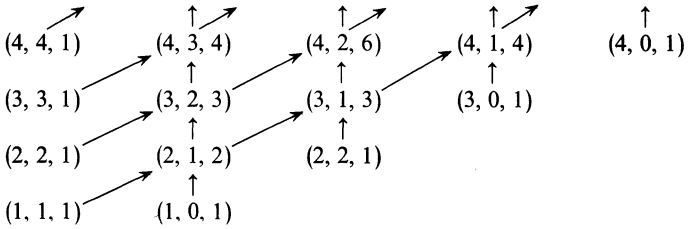


**Figure 1.**
**Le triangle de Pascal.**

Nous définirons formellement les calculs ascendants, sous-entendant par algorithmes ascendants les moyens de construire ces calculs. Les algorithmes ascendants manipulent des ensembles de points du graphe, et utilisent un mécanisme de production : un ensemble de points peut produire un nouveau point s'il contient tous ses précécesseurs immédiats. Ces algorithmes sont aussi appelés jeux de pions dans la littérature : les ensembles de points sont alors représentés par des ensembles de pions progressant sur le graphe (voir [7, 9, 13]).

Les avantages des algorithmes ascendants sont clairs : gain de temps si l'on peut supprimer la gestion des piles et éviter de recalculer des valeurs intermédiaires; gain de mémoire si l'on peut supprimer les piles et décider quand une valeur intermédiaire ne servira plus. Il n'existe cependant pas de moyen systématique pour construire des algorithmes ascendants efficaces. Il faut tout d'abord caractériser la structure de récursion associée au programme et déterminer le domaine de chaque point, pour éviter de calculer des points inutiles (points n'appartenant pas au domaine du point cherché). Ensuite, il faut découvrir un algorithme, montrer sa terminaison et sa cor-

rection. En [3, 4], nous donnons quelques outils permettant d'aborder ces problèmes. Nous devons toutefois souligner que les réécritures restent la seule méthode de calcul générale.

## I.2. Définition formelle des calculs ascendants. Optimalité

Nous supposons la structure de récursion connue par la donnée de $mp(x)$ et $dom(x)$ pour tout point $x$.

DÉFINITIONS : Un *calcul ascendant*, ou *séquence de configuration* de *mémoire* (en abrégé *s.c.m.*), est une suite $Y = Y_0, Y_1, \ldots, Y_m$ d'ensemble finis de points du graphe vérifiant :

(i) $Y_0 = \emptyset$

(ii) $\forall i \quad 1 \leqslant i \leqslant m, \qquad Y_i \subset Y_{i-1} \cup \{ x \mid mp(x) \quad Y_{i-1} \}$

(partant de l'information vide, on peut à chaque étape produire de nouveaux points et se débarrasser d'anciens points devenus inutiles).

*Une s.c.m. $Y$ calcule $x$* s'il existe $m$ tel que $x \in Y_m$.

Nous utiliserons aussi des *suites ascendantes*, qui sont des suites $X = x_1, x_2, \ldots, x_m$ telles que pour tout $i \geqslant 1$ on a $mp(x_i) \subset \{ x_j \mid j < i \}$. Nous noterons alors $X_i = \{ x_j \mid j \leqslant i \}$. Une s.c.m. $Y$ est dite *compatible avec la suite $X$* si elle vérifie

$$\forall i \geqslant 1, \quad x_i \in Y_i \subset X_i \qquad \text{et} \qquad mp(x_{i+1}) \subset Y_i.$$

Ceci exprime que la s.c.m. $Y$ est une implantation possible de la suite $X$. Remarquons que la suite des $X_i$ est une s.c.m. compatible avec $X$ : nous l'appellerons s.c.m. canoniquement associée à $X$.

Le *temps de calcul* d'une s.c.m. $Y = Y_0, Y_1, \ldots, Y_n$ est le nombre de productions qu'elle effectue, un même point étant compté à chaque fois qu'il est produit :

$$T(Y) = \sum_{i=1}^{m} \text{cardinal}\,(\{ y \in Y_i \mid y \notin Y_{i-1} \})$$

La *mémoire* utilisée par $Y$ est la taille maximale des $Y_n$ :

$$M(Y) = \max_{1 \leqslant k \leqslant n} \text{cardinal}\,(Y_k)$$

Une s.c.m. calculant $x$ est *optimale en temps* (resp. *optimale en mémoire*) pour le calcul de $x$ si elle utilise moins de temps (resp. mémoire) que toute autre s.c.m. calculant $x$. Notons que toutes les s.c.m. optimales en temps calculent une fois et une seule tous les points de $dom(x)$, et eux seulement, et que les optimalités en temps et en mémoire sont en général contradictoires (voir [9]). La cons-

truction de calculs optimaux rejoint les problèmes d'allocation de registres [16] et de jeux de pions [7, 9, 13].

Insistons encore sur le fait qu'un résultat d'optimalité n'est valable que vis-à-vis de la structure de récursion donnée, et non vis-à-vis de la fonction calculée : cette fonction peut éventuellement être définie par d'autres structures de récursion plus efficaces. De même nous ne définissons que le coût d'un calcul et non celui d'un algorithme ou d'un programme.

### I.3. Coûts de l'appel par valeur

Dans notre classification des modes de calcul, l'appel par valeur joue un rôle hybride. C'est une règle de réécriture, mais c'est aussi un algorithme d'exploration alternativement descendante et ascendante de la structure de récursion : les empilages correspondent à une « construction descendante » de la structure et des dépilages aux calculs ascendants proprement dits. La *suite ascendante calculée par l'appel par valeur* pour $c(n, p)$ est par exemple écrite par le programme

*fonction* $c(n, p)$ : *si* $p = 0$ *ou* $p = n$ *alors* $c \leftarrow 1$ ; écrire $(n, p, 1)$
*sinon* $c \leftarrow c(n-1, p) + c(n-1, p-1)$;
écrire $(n, p, c)$.

Le calcul ascendant ainsi défini n'est en général pas optimal en temps : ici $(n-2, p-1, c(n-2, p-1))$ est calculé deux fois. En revanche tous les points calculés sont utiles, c'est-à-dire appartiennent à $dom(n, p, c(n, p))$ : ceci est vrai dès que l'appel par valeur est une « règle de réécriture optimale » au sens de [5, 17], et en particulier dès que la fonction calculée est stricte (indéfinie dès qu'un argument est indéfini, voir [17]). Nous obtenons dans ce cas une *évaluation de l'appel par valeur en tant que règle de réécriture* pour le calcul d'un point $x$ :

le *nombre de réécritures* effectué par l'appel par valeur est égal au nombre de points de la suite ascendante qu'il engendre, puisque nous produisons un point par réécriture. Le coût en temps de l'appel par valeur est donc le même qu'on le considère comme règle de réécriture ou comme algorithme ascendant. L'appel par valeur développant le graphe $dom(x)$ en un arbre, son coût est la taille de cet arbre, aussi égal au nombre de chemins distincts vers $x$ dans la structure de récursion.

la *taille maximale de la pile* dans une implantation classique est la longueur du plus long chemin vers $x$ dans la structure de récursion.

REMARQUE : Il est couramment admis que l'inconvénient majeur de l'appel par valeur est d'être incorrect, comme le montre l'exemple de Morris [11] :

$$f(x, y) = \textit{si } x = 0 \textit{ alors } 0 \textit{ sinon } f(x-1, f(x, y))$$

Ici $f(2, 1)$ boucle en appel par valeur, et s'évalue en 0 en appel par nom. Cependant *même lorsqu'il est correct, l'appel par valeur peut ne pas être optimal,* comme le montre l'exemple suivant :

$$f(x, y) = \textit{si } x = 0 \textit{ alors } 0 \textit{ sinon } f(x - 1, f(x - 1, y))$$

Ici l'appel par valeur est correct mais moins efficace que l'appel par nom. Nos évaluations restent valides en modifiant la définition [3. 4] pour définir la structure de récursion associée à l'appel par valeur, voir [12]. Mais cette structure est alors moins efficace pour calculer la fonction que la structure de récursion que nous considérons en [3, 4], liée à l'appel par nom (ou à l'appel par valeur lorsqu'il est optimal).

REMARQUE : Pour éviter les recalculs, on peut garder en mémoire les valeurs précédemment calculées, ce qui revient à implémenter la s.c.m. canoniquement associée à l'appel par valeur. Elle est optimale en temps, mais utilise cardinal (dom($x$)) mémoires, ce qui est le plus souvent prohibitif. Si on cherche à optimiser cette s.c.m. en se débarrassant dès que possible des valeurs inutiles, on n'obtient pas toujours une s.c.m. optimale en mémoire (voir [4]).

## II. ANALYSE DU PROGRAMME DE J. ARSAC COMPARAISON AVEC CELUI DE RICE

### II.1. Analyse du programme

Nous analyserons le programme suivant, obtenu par quelques transformations élémentaires de P10 dans [1] : suppression de $c[1]$ toujours nul, traitement séparé des cas $u = 0$ et $u = 1$ ...

**PR1 :** SI $m = 0$ ALORS $v \leftarrow n + 1$ SINON SI $m = 1$ ALORS $v \leftarrow n + 2$ SINON
$u \leftarrow m$; $v \leftarrow n$; $c[2{:}m] \leftarrow 0$
{ SI $u = 1$ ALORS $v \leftarrow v + 2$; $u \leftarrow 2$
SINON $c[u] \leftarrow v$; $c[2{:}u - 1] \leftarrow 1$; $v \leftarrow 3$; $u \leftarrow 2$ IS;
{ SI $u > m$ ALORS !2 SINON SI $c[u] \neq 0$ ALORS !
SINON $u \leftarrow u + 1$ IS IS };
$c[u] \leftarrow c[u] - 1$; $u \leftarrow u - 1$ } IS IS

Les données sont $m$ et $n$ et le résultat est $v$. Ce programme a été obtenu à partir du programme récursif suivant :

**PR2 :** $Ack(m, n) \leftarrow$ SI $m = 0$ ALORS $n + 1$ SINON SI $n = -1$ ALORS 1
SINON $Ack(m - 1, Ack(m, n - 1))$ IS IS

REMARQUE : PR2 ne diffère que très peu du programme que nous avons donné dans l'introduction : poser $Ack(m, -1) = 1$ simplifie la présentation récursive en modifiant de façon mineure la structure de récursion : les

points $(m, o, z)$ ont maintenant pour prédécesseurs immédiats $(m, -1, 1)$ et $(m-1, 1, z)$ au lieu de n'avoir que $(m-1, 1, z)$; cette modification était implicitement utilisée par Rice [14].

On voit que PR1 calcule directement $Ack(1, n) = n + 2$. Ceci correspond pour PR2 à une optimisation évidente :

**PR3** : $Ack(m, n) \leftarrow$ SI $m = 0$ ALORS $n + 1$ SINON SI $m = 1$ ALORS $n + 2$
SINON SI $n = -1$ ALORS 1
SINON $Ack(m - 1, Ack(m, n - 1))$ IS IS IS

(On pourrait aussi calculer directement $Ack(2, n) = 2n + 3$, puis $Ack(3, n) = 2^{n+3} - 3$ etc...). Nous omettrons désormais les cas triviaux $m = 0$ et $m = 1$, en supposant $m \geqslant 2$.

La structure de récursion associée à PR3 est montrée figure 2. Elle est donnée par $mp(1, n, n + 2) = mp(m, -1, 1) = \varnothing$ et pour $m > 1$, $n \geqslant 0$ :

$mp(m, n, Ack(m, n))$
$= \{(m, n - 1, Ack(m, n - 1)), (m - 1, Ack(m, n - 1), Ack(m, n))\}$.

On montre comme en [4] que toute s.c.m. calculant $(m, n, A(m, n))$ utilise au moins $m$ mémoires.

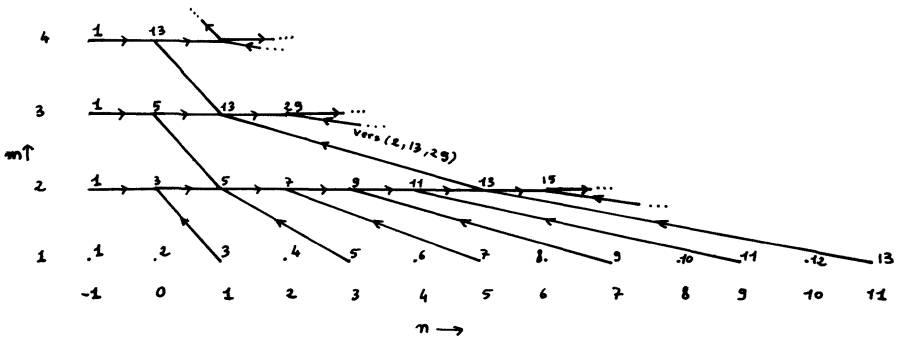


**Figure 2.**
**La structure de récursion associée à PR3. Ici $m$ est indiqué en ordonnées, $n$ en abscisse, et $Ack(m, n)$ est porté sur le point correspondant.**

La séquence ascendante associée à l'appel par valeur pour $Ack(3, 2)$ est montrée figure 3. Elle ne calcule pas de points inutiles, mais effectue des recalculs et n'est donc pas optimale en temps. Nous allons montrer informellement que *PR1 calcule la même séquence ascendante en utilisant m mémoires* (abstraites) : PR1 définit une s.c.m. optimale en mémoire mais non en temps. Une méthode systématique pour vérifier cette hypothèse serait d'insérer des

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| * | 1 | 1 | 3 | | 1 | 7 | 9 |
| * | 2 | 0 | 3 | | 2 | 3 | 9 |
| * | 1 | 3 | 5 | | 1 | 9 | 11 |
| * | 2 | 1 | 5 | | 2 | 4 | 11 |
| * | 3 | 0 | 5 | | 1 | 11 | 13 |
| | 1 | 1 | 3 | | 2 | 5 | 13 |
| | 2 | 0 | 3 | * | 1 | 13 | 15 |
| | 1 | 3 | 5 | * | 2 | 6 | 15 |
| | 2 | 1 | 5 | * | 1 | 15 | 17 |
| * | 1 | 5 | 7 | * | 2 | 7 | 17 |
| * | 2 | 2 | 7 | * | 1 | 17 | 19 |
| * | 1 | 7 | 9 | * | 2 | 8 | 19 |
| * | 2 | 3 | 9 | * | 1 | 19 | 21 |
| * | 1 | 9 | 11 | * | 2 | 9 | 21 |
| * | 2 | 4 | 11 | * | 1 | 21 | 23 |
| * | 1 | 11 | 13 | * | 2 | 10 | 23 |
| * | 2 | 5 | 13 | * | 1 | 23 | 25 |
| * | 3 | 1 | 13 | * | 2 | 11 | 25 |
| | 1 | 1 | 3 | * | 1 | 25 | 27 |
| | 2 | 0 | 3 | * | 2 | 12 | 27 |
| | 1 | 3 | 5 | * | 1 | 27 | 29 |
| | 2 | 1 | 5 | * | 2 | 13 | 29 |
| | 1 | 5 | 7 | * | 3 | 2 | 29 |
| | 2 | 2 | 7 | | | | |

**Figure 3.**

**La suite ascendante de l'appel par valeur pour $Ack(3, 2)$ (lire de haut en bas et de gauche à droite). Marquée *, la suite ascendante de Rice ; non marqués, les recalculs de l'appel par valeur.**

« instructions de production » de la séquence ascendante et de les suivre au cours des transformations. Nous indiquerons seulement ici les effets des transformations :

**P1 → P4 :** simulation de l'appel par valeur à l'aide d'une pile pour $u$ ; $v$ n'a pas besoin d'être empilé. Il y a en fait trois opérations de production implicite dans P4, que nous explicitons ici en italiques :

$v \leftarrow$ SI $u = 0$ ALORS $v + 1$ ; *produire* $(o, v, v + 1)$
SINON 1 ; *produire* $(u, -1, 1)$ IS

et plus loin :

SINON *dep*$(u)$ ; *produire* $(u, u', v)$ ; $u \leftarrow u - 1$
où $u'$ est le seul entier tel que $Ack\ (u, u') = v$

**P4 → P6 :** codage de la pile par le vecteur $c$ ; l'instruction *dep*$(u)$ est remplacée par $c[u] \leftarrow c[u] - 1$.

**P6 → P7 :** suppression de la boucle de tête, qui était une boucle d'empilages : c'est la partie cruciale de l'optimisation en mémoire du calcul ascendant : elle correspond à *la destruction dans la configuration de mémoire de tous les points précédemment calculés d'abscisse inférieure ou égale à u*, car on exécute en fait $c[u] \leftarrow c[u] + v$ suivi de

$c[1{:}u-1] \leftarrow 1$. Cette transformation est possible du fait qu'on a pas eu besoin d'empiler $v$ dans $P$. Les instructions liées à la production n'étant pas modifiées, la suite ascendante ne l'est pas non plus.

**P7 → P10 :** traitement direct du cas $u = 1$, donc passage de la structure de récursion de PR2 à celle de PR3.

Finalement, le programme PR1 construit donc une s.c.m. compatible avec la suite ascendante de l'appel par valeur et utilisant $m$ mémoires, une par valeur de la première composante des triplets. On peut mettre directement en évidence cette s.c.m. dans PR1, au moyen d'un tableau $Y[1{:}m;\ 1{:}3]$ où $Y[u;]$ contient un triplet de première composante $u$ :

**PR4 :** $u \leftarrow m$; $v \leftarrow n$; tableaux $c[2{:}m]$, $Y[1{:}m;\ 1{:}3]$;
{ SI $u = 1$ ALORS $Y[1;] \leftarrow (1, v, v+2)$; $v \leftarrow v+2$
SINON $c[u] \leftarrow v$; $c[2{:}u-1] \leftarrow 1$; $v \leftarrow 3$;
$Y[1;] \leftarrow (1, 1, 3)$
*POUR i DE 2 A u FAIRE* $Y[i;] \leftarrow (i, -1, 1)$ *FPOUR*
IS IS;
$u \leftarrow 2$;
{ SI $u > m$ ALORS !2 SINON $Y[u;] \leftarrow (u, Y[u;2]+1, v)$;
SI $c[u] \neq 0$ ALORS ! SINON $u \leftarrow u+1$ IS IS }
$c[u] \leftarrow c[u]-1$; $u \leftarrow u-1$ }

Notons que PR4 s'obtient à partir de PR1 en ajoutant des instructions inutiles. On voit alors que « chercher le plus petit $u$ tel que $c[u] \neq 0$ » revient à « produire tous les points de dom$(m, n, Ack(m, n))$ ayant pour troisième composante $v$ ». En fait on a $c[m] = 0$ si et seulement si $Y[m;2] = n$, et pour $u < m$, $c[u] = 0$ si et seulement si $Y[m+1;2] = Y[m;3]$, c'est-à-dire s'il est possible de produire un nouveau point d'abscisse $m+1$ à partir de $Y[m;]$ et $Y[m+1;]$ : pour $u < m$, $c[u]$ *est le nombre de points qu'il faut produire à l'abscisse u avant de pouvoir produire un nouveau point à l'abscisse* $u+1$. On voit bien comment sont effectués les recalculs : lorsque la boucle intérieure se termine avec $u > 2$, on fait $c[u] \leftarrow v$ et $c[2{:}u-1] \leftarrow 1$, ce qui revient à réinitialiser le programme pour le calcul de $(u, v, Ack(u, v))$ en perdant toute l'information acquise sur les points d'abscisse inférieure ou égale à $u$.

## II.2. L'algorithme de Rice

Nous définissons la s.c.m. de Rice à l'aide d'une suite ascendante $d_k = (m_k, n_k, z_k)$, en utilisant un tableau $Y[1 : m]$ dont les états successifs sont les ensembles de la s.c.m. :

(i) initialisation : pour $2 \leqslant i \leqslant m$, $Y[i] \leftarrow (i, -1, 1)$.
(ii) $d_1 = (1, 1, 3)$ et $Y[1] \leftarrow d_1$.

(iii) si $d_k = (m_k, n_k, z_k)$ avec $m_k = m$ et $n_k = n$, *alors stop.*

(iv) sinon si $d_k = (m_k, n_k, z_k)$ avec $m_k < m$ et si
$Y[m_k + 1] = (m_k + 1, n', n_k)$, alors $d_{k+1} = (m_k + 1, n' + 1, z_k)$
et $Y[m_k + 1] \leftarrow d_{k+1}$.

(v) sinon si $d_k = (m_k, n_k, z_k)$ alors $d_{k+1} = (1, z_k, z_k + 2)$.

Cette s.c.m. n'effectue ni calculs inutiles, ni recalculs : elle est optimale en temps par le calcul de $(m, n, Ack(m, n))$. Utilisant $m$ mémoires, elle est optimale en mémoire. L'ordre d'introduction des points est l'ordre d'introduction des nouveaux points dans l'appel par valeur (*cf. fig.* 3) : pour chaque nouvelle valeur de $z$, on introduit tous les points $(m', n', z)$ du graphe vérifiant $1 \leqslant m' \leqslant m$, dans l'ordre des $m'$ croissants. On peut en fait programmer cet algorithme en utilisant le vecteur $c$ de PR4 :

```
PR5 : tableaux c[2:m], Y2[2:m], Y[1:m; 1:3];
      c[m] ← n; v ← 1
      POUR i DE 2 A M FAIRE Y2[i] ← − 1; c[i] ← 1;
                            Y[i;] ← (i, − 1, 1)
      { Y[1] ← (1, v, v + 2); v ← v + 2; u ← 2;
        SI u > m ALORS !2 SINON Y2[u] ← Y2[u] + 1;
                                Y[u;] ← (u, Y[u;2] + 1, v)
                SI c[u] ≠ 0 ALORS ! SINON c[u] ← v − Y2[u] − 1;
                   u ← v + 1 IS IS }
                   c[u] ← c[u] − 1 }
```

Comme dans PR4, les instructions concernant $Y$ sont inutiles et servent à illustrer la formation de la s.c.m.

La différence essentielle entre PR5 et PR4 est que le vecteur $c$ est *réactualisé* par $c[u] \leftarrow v - Y2[u] - 1$ au lieu d'être réinitialisé. Ce gain en temps s'accompagne d'une *perte en mémoire au niveau du programme*, puisqu'il faut conserver le vecteur $Y2$, qui est en fait le vecteur des secondes composantes de $Y$ : le programme de Rice utilise deux mémoires par mémoire théorique, alors que celui d'Arsac n'en utilise qu'une, comme nous l'avions signalé dans l'introduction.

Comparons maintenant les *temps de calcul* des deux calculs ascendants : par une récurrence élémentaire (voir [4]), on voit que le temps de calcul de la s.c.m. associée à PR4 est donné par les équations de récurrence :

(i) $t4(1, n) = t4(m, -1) = 1$

(ii) $t4(m, n) = 1 + t4(m, n - 1) + t4(m - 1, Ack(m, n - 1))$
pour $m > 1$ et $n \geqslant 0$

Le temps de calcul de la s.c.m. associée à PR5 est la taille de dom $(m, n, Ack\ (m.\ n))$. donnée simplement par :

(i) $t5(1, n) = t5(m, -1) = 1$

(ii) $t5(m, n) = n + 2 + t5(m - 1, Ack(m, n - 1))$

pour $m > 1$ et $n \geqslant 0$

Sans résoudre ces équations, on peut voir que $t4(m, n)$ et $t5(m, n)$ sont tous les deux asymptotiquement linéaires en $Ack(m, n)$, ce qui justifie ce que nous avons annoncé dans l'introduction : les recalculs ne conduisent pas ici à une explosion exponentielle du temps de calcul.

## CONCLUSION

J. Arsac utilise en [1] trois types de transformations de programmes itératifs : transformations syntaxiques, sémantiques locales, sémantiques profondes. Le statut des transformations syntaxiques et sémantiques locales est clair : ce sont celles que peut réaliser un système évolué de manipulation de programmes [10] : modification de la présentation du programme, reconnaissance de schémas-type et application des transformations correspondantes, modifications locales de la séquence des calculs par utilisation de propriétés élémentaires des opérations et de l'affectation. Les optimisations du temps de calcul réalisées par ces transformations sont le plus souvent linéaires ce qui est important en pratique. Notons que dans certains cas des transformations syntaxiques simples des programmes récursifs peuvent aussi conduire à une optimisation exponentielle du temps de calcul, par élimination des recalculs (voir [6]).

En revanche, les transformations sémantiques profondes sont fondées sur la mise en évidence de propriétés globales de la suite des calculs, des objets manipulés ou de la fonction calculée. Elles ont pour but des améliorations profondes du programme, et en ce sens peuvent être vues aussi comme des transformations profondes de l'« algorithme » que code le programme. Nous pensons que ces transformations doivent le plus souvent être fondées sur une analyse d'algorithmes, et qu'il existe d'autres représentations des algorithmes qui sont beaucoup plus adaptées à l'analyse que les programmes itératifs. En effet l'écriture d'un programme itératif classique suppose effectués deux choix : choix de l'ordre des calculs, choix de la représentation des objets. Or ces choix sont précisément ceux que l'analyse d'algorithme a pour but de guider, ce qui signifie qu'il est préférable de les retarder au maximum. De plus certaines propriétés immédiates sur la définition initiale peuvent devenir difficiles à voir après codage.

Enfin, lorsqu'une transformation de programme réalise une optimisation importante, c'est probablement parce qu'on a mis en évidence une propriété

importante de l'algorithme, qu'on peut peut-être exploiter plus directement. Illustrons ces points à l'aide de l'exemple d'Ackermann :

– Le choix initial de J. Arsac est de transformer immédiatement le programme récursif en programme itératif. Il conduit à adopter la stratégie de l'appel par valeur et à introduire une pile. Dans les transformations suivantes, la pile disparaît mais les recalculs sont conservés. Rappelons les inconvénients de l'appel par valeur : il engendre des recalculs et ne conduit pas toujours à une bonne gestion mémoire, même après optimisation. Il existe aussi des cas où il est préférable de calculer certaines valeurs intermédiaires inutiles que de chercher à les éliminer (voir partitions d'un nombre entier [2]). Une telle stratégie est probablement difficile à déduire de celle de l'appel par valeur, qui ne calcule que des valeurs intermédiaires utiles.

– L'amélioration $Ack(1, n) = n + 2$, qui est évidente sur la formulation récursive, nécessite plusieurs transformations. Une amélioration comme $Ack\ (3, n) = 2^{n+3} - 3$ serait certainement plus difficile à voir.

– Une fois la pile codée, on s'aperçoit que les éléments du tableau $c$ sont réinitialisés avec des valeurs de la fonction. Ceci suggère qu'il y a une relation forte entre les calculs de la fonction et ses valeurs. Mais cette information est difficile à exploiter directement, car on ne sait plus pour quels arguments la valeur a été calculée.

Nous proposons en [3, 4] une méthode d'analyse d'algorithmes définis par programmes récursifs. La première étape n'est plus le choix d'un codage, mais la détermination d'une structure de récursion, qui correspond à toute une famille d'algorithmes. Ceux-ci peuvent être exprimés, transformés et comparés dans le formalisme des calculs ascendants qui repose sur la seule opération de production. Dans le cas du programme d'Ackermann, la détermination de la structure de récursion, la conception de l'algorithme de Rice et la preuve de sa correction et de son optimalité en temps n'offrent pas de difficultés ; la preuve de son optimalité en mémoire repose sur des techniques connues [7]. Nos méthodes semblent cependant ne s'appliquer commodément qu'à des programmes manipulant des structures de données simples. D'autres techniques similaires d'analyse et transformation simultanée d'algorythmes peuvent être employées lorsqu'on manipule des objets complexes, comme des ensembles ou des graphes, voir [8, 18].

## BIBLIOGRAPHIE

1. J. ARSAC, *Emploi de méthodes constructives en programmation : un dossier la fonction d'Ackermann*, R.A.I.R.O., ce même numéro, article précédant celui-ci.
2. D. W. BARRON, *Techniques récursives en programmation*, Dunod, Paris, 1970.

3. G. BERRY, *Bottom-up Computations of Recursive Programs.* R.A.I.R.O., **10**, n° 3, mars 1976, p. 47 à 82.
4. G. BERRY, *Calculs ascendants des Programmes Récursifs*, Thèse de 3e cycle, Université, Paris VII, Paris, avril 1976.
5. G. BERRY et J. J. LEVY, *Minimal and Optimal Computations of Recursive Programs.* Proc. 4 th Annual ACM SIGACT-SIGPLAN Conference on Principles of Programming Languages, Los Angeles, California, janvier 1977, p. 215-226.
6. R. BURSTALL et J. DARLINGTON, *A Transformation System for Developing Recursive Programs*, Department of Artificial Intelligence, University of Edinburgh, U.K., Research Report No. 19, 1976.
7. S. A. COOK, *An observation of Time-Storage Trade-off.* Proc. 5 th annual ACM Symposium on Theory of Computation, Austin, Texas, 1973, p. 29-33.
8. P. FLAJOLET, *Algorithmes d'Exploration d'Arbres*, A paraître en Rapport Laboria, IRIA.
9. J. HOPCROFT, W. PAUL et L. VALIANT, *On time versus Space and Related Problems.* Proc. 16 th Annual Symposium on Foundations of Computer Science, Berkeley, California, 1975, p. 57-64.
10. G. HUET, G. KAHN et al., *MENTOR, Système d'édition et de transformation de programmes.* A paraître en Rapport Laboria, IRIA.
11. J. MORRIS, *Another Recursion Induction Principle.* Comm. ACM, **14**, 1971, p. 351-354.
12. D. PARK, *Fixpoint Induction and Proof of Program Properties.* Machine Intelligence 5, Edinburgh University Press, 1969, p. 59-77.
13. W. PAUL, R. TARJAN et J. CELONI, *Space bounds for a game of pebbles.* Proc 8th Annual ACM Symposium on Theory of Computing, Hershey, Pennsylvania, 1976, p. 149-160.
14. H. G. RICE, *Recursion and Iteration*, Comm. ACM, **8**, 1965, p. 114-115.
15. D. W. RITCHIE, *Program structure and Computational Complexity.* Ph. D. Thesis, Harvard, 1967.
16. R. SETHI et J. D. ULLMAN, *The generation of Optimal Code for Arithmetic Expressions.* Journal of ACM, **17**, 1970, p. 715-728.
17. J. VUILLEMIN, *Syntaxe, Sémantique et Axiomatique d'un Langage de Programmation simple*, Thèse de doctorat ès Sciences Mathématiques, Université Paris VI, Paris, 1974.
18. J. VUILLEMIN, *Notes de cours*, Université d'Orsay, 1976.
19. A. YASUHARA, *Recursive Functions theory and Logic*, Academic Press, New York, 1971.