

JACQUES ARSAC

**La fonction d’Ackermann : un nouveau
mode de dérécursivation**

Informatique théorique et applications, tome 20, n° 2 (1986),
p. 149-156

http://www.numdam.org/item?id=ITA_1986__20_2_149_0

© AFCET, 1986, tous droits réservés.

L'accès aux archives de la revue « Informatique théorique et applications » implique l'accord avec les conditions générales d'utilisation (<http://www.numdam.org/conditions>). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques

<http://www.numdam.org/>

LA FONCTION D'ACKERMANN : UN NOUVEAU MODE DE DÉRÉCURSIVATION (*)

par Jacques ARSAC (1)

Communiqué par J. BERSTEL

Résumé. – Nous avons présenté en 1977 une façon de déduire de la définition récursive de la fonction d'Ackermann une procédure itérative nécessitant moins de mémoire que la procédure de Rice. Les progrès réalisés en matière de transformation de la récursivité en itération permettent maintenant de retrouver le résultat de façon extrêmement simple. Cet exemple illustre le fait qu'il semble préférable de chercher une nouvelle forme récursive pour faciliter le passage à l'itération.

Abstract. – In 1977, we have published the computations by which the recursive definition of Ackermann's function can be transformed into an iterative form, requiring less storage than Rice's one. Methods for transforming recursion into iteration have been improved, so that the same result can be obtained now without sophisticated computations. We show on this example that it is better to transform first the recursive form into another one, better fitted for transformation into iteration.

1. INTRODUCTION

La transformation de la récursivité en itération a été abondamment étudiée, tant du point de vue théorique [9, 10, 13] que du point de vue pratique [1, 3, 4]. Des systèmes automatiques ont été réalisés [6, 7, 12]. Mais ces derniers ne sont vraiment efficaces que pour des schémas monadiques pas trop complexes, et éventuellement un très petit nombre de schémas dyadiques (généralisant plus ou moins le schéma de Fibonacci [6]). Les méthodes qui ont été proposées pour l'emploi des transformations dans une méthodologie de programmation qui cherche à déduire les programmes itératifs de leurs définitions récursives prises comme « axiomes » [1, 3, 4] élargissent le domaine d'application de ce type de transformation : dans la mesure où l'on se réfère à une méthode, on ne souffre plus autant des limitations imposées par tout catalogue de schémas, nécessairement fortement borné. Mais les définitions récursives dyadiques continuent à poser de sérieuses difficultés. Il faut, d'une façon quelconque, les ramener à une définition monadique, ce qui peut

(*) Reçu novembre 1984, révisé février 1985.

(1) École Normale Supérieure, 45, rue d'Ulm, 75005 Paris.

demander un gros effort d'imagination, le fameux « Eureka » selon Burstall [5, 6].

Dans une première approche, nous avons proposé de remplacer une définition dyadique de fonction, telle celle d'Ackermann, par une procédure récursive comportant un appel récursif dans une boucle. On possède les outils permettant la transformation de celle-ci en procédure itérative [1]. Mais les calculs peuvent être très complexes. C'est ce qui apparaît clairement dans la publication que nous avons faite en 1977 de l'application de cette méthode de transformation à la fonction d'Ackermann [2].

Or, comme Burstall l'avait déjà suggéré en 1977 [7], il apparaît aujourd'hui qu'il est bien préférable de commencer par modifier la définition récursive, par des transformations faites au niveau récursif et utilisant les outils classiques de généralisation et de substitution (folding-unfolding) [3, 6, 7]. Ce point a fait l'objet d'une récente thèse [8]. On manque encore de repères permettant de guider la transformation à faire au niveau récursif en vue d'obtenir une forme facile à transformer, si tant est que de tels guides puissent exister [16]. Nous allons montrer ici comment un choix judicieux de transformation de la définition récursive dyadique de la fonction d'Ackermann en définition monadique permet de retrouver facilement les résultats déjà publiés [2].

2. PASSAGE A UNE DÉFINITION RÉCURSIVE MONADIQUE

Rappelons d'abord la définition de la fonction d'Ackermann. m et n désignant des entiers naturels :

$$A(m, n) = \text{SI } m=0 \text{ ALORS } n+1 \\ \text{SINON SI } n=0 \text{ ALORS } A(m-1, 1) \\ \text{SINON } A(m-1, A(m, n-1)) \\ \text{IS} \\ \text{IS}$$

Pour éviter d'avoir un appel monadique dans une branche du SI, un appel dyadique dans l'autre, nous étendons le domaine de définition au cas $n = -1$:

$$A(m, n) = \text{SI } m=0 \text{ ALORS } n+1 \\ \text{SINON SI } n=-1 \text{ ALORS } 1 \\ \text{SINON } A(m-1, A(m, n-1)) \\ \text{IS} \\ \text{IS}$$

Nous voulons obtenir une définition récursive monadique. La difficulté vient ici du mécanisme d'emboîtement des appels. Remplaçons l'appel intérieur dans la définition ci-dessus par sa valeur, tirée de cette même définition :

$$A(m, n) = A(m-1, A(m-1, A(m, n-2))).$$

Dans le cas général, nous allons avoir une suite de A emboîtés en arguments droits : il apparaîtra ainsi une suite d'arguments gauches, mais un seul argument droit :

$$A(u_1, A(u_2, \dots, A(u_p, v) \dots)).$$

Nous voulons faire de ceci une unique fonction f . Pour qu'elle représente cette suite d'appels emboîtés, donnons lui comme argument gauche la suite des u , et pour argument droit l'unique argument droit v . Notons $s = u_1 : u_2 : \dots : u_p$ la suite des u :

$$f(s, v) = f(u_1 : u_2 : \dots : u_p, v) = A(u_1, A(u_2, \dots, A(u_p, v) \dots)).$$

Nous pouvons tout aussi bien considérer qu'il n'y a que $p-1$ emboîtements de fonctions A , la dernière ayant pour argument droit non pas v mais $A(u_p, v)$:

$$f(u_1 : \dots : u_{p-1} : u_p, v) = f(u_1 : \dots : u_{p-1}, A(u_p, v)).$$

Nous avons ainsi obtenu une fonction récursive monadique que nous allons pouvoir définir complètement par substitution.

Insistons bien : l'étape créative, l'effort d'imagination, le Eureka selon Burstall réside dans la décision de remplacer A par une fonction f ayant pour argument gauche la suite des u ; tout le reste ne sera maintenant que manipulation algébrique simple. Précisons d'abord la définition de f . Nous avons besoin d'isoler le dernier terme d'une suite. Nous noterons $\text{der}(s)$ le dernier terme de s , et $\text{deb}(s)$ la suite s privée de son dernier terme. La suite vide sera notée nil :

$$\text{der}(s : u) = u, \quad \text{deb}(s : u) = s, \quad s = \text{deb}(s) : \text{der}(s)$$

Suivant ce que l'on a déjà obtenu :

$$f(s, v) = f(\text{deb}(s), A(\text{der}(s), v)),$$

$$f(\text{nil} : u, v) = A(u, v),$$

$$f(\text{nil}, v) = v.$$

Utilisons les valeurs connues de A lorsqu'il n'y a pas récursivité :

$$f(s : 0, v) = f(s, A(0, v)) = f(s, v + 1),$$

$$f(s : u, -1) = f(s, A(u, -1)) = f(s, 1) \quad (u \neq 0).$$

Utilisons maintenant la partie récursive de la définition de A :

$$f(s : u, v) = f(s, A(u, v)) = f(s, A(u-1, A(u, v-1)))$$

$$= f(s : (u-1) : u, v-1).$$

La fonction f est maintenant complètement définie par une définition récursive, monadique, terminale, donnant trivialement un programme itératif calculant A . Mais parce qu'il y a intérêt à travailler le plus longtemps possible au niveau récursif, nous allons d'abord utiliser les propriétés ci-dessus pour condenser les opérations.

Dans

$$f(s : u, v) = f(s : (u-1) : u, v-1)$$

nous pouvons à nouveau insérer dans la suite un terme $u-1$ et diminuer v de 1. Nous arriverons ainsi à :

$$f(s : u, v) = f(s : (u-1)^{v+1} : u, -1) = f(s : (u-1)^{v+1}, 1)$$

où $s : x^y$ dénote la concaténation à s de y termes égaux à x . Appliquons cette relation avec $v=1$:

$$\begin{aligned} f(s : u, 1) &= f(s : (u-1) : (u-1), 1) \\ &= f(s : (u-1) : (u-2) : \dots : 1 : 0 : 0, 1). \end{aligned}$$

Comme :

$$\begin{aligned} f(s : 0, v) &= f(s, v+1), \\ f(s : 0^x, v) &= f(s, v+x). \end{aligned}$$

Nous avons tout ce qui est nécessaire pour calculer A .

3. PASSAGE A L'ITÉRATION

Suivant la méthode développée en [2], nous construisons le programme itératif à partir de l'invariant de boucle, lui-même tiré de la définition récursive de f . Comme celle-ci est récursive terminale, et qu'il y a intérêt à faire apparaître le dernier terme de la suite, argument gauche de f , nous poserons :

$$\llbracket \text{résultat} = f(s : u, v) \rrbracket$$

en supposant s, u, v connus.

$$\text{Si } u \neq 0, f(s : u, v) = f(s : (u-1)^v : (u-2) : \dots : 1 : 0 : 0, 1).$$

On obtient :

$$s : = s : (u-1)^v : (u-2) : \dots : 1 : 0; u : = 0; v : = 1.$$

Nous obtenons ainsi un morceau de programme avec ses assertions :

```

[[résultat = f(s : u, v)]]
SI u ≠ 0 ALORS s := s : (u-1) : (u-2) : ... : 1 : 0; u := 0; v := 1 IS
[[résultat = f(s : u, v) ET u = 0]].

```

Faisons apparaître les zéros à droite de s $s = s' : 0^x$:

$$f(s : u, v) = f(s' : 0^{x+1}, v) = f(s', v + x + 1)$$

Si s' est vide, $f(\text{nil}, v + x + 1) = v + x + 1$ et c'est fini. Sinon, on retrouve l'invariant de boucle en prenant $u = \text{der}(s')$ et $s = \text{deb}(s')$.

Pour démarrer, partant de $\text{résultat} = A(m, n) = f(\text{nil} : m, n)$ il suffit de prendre $s = \text{nil}$ $u = m$ et $v = n$. D'où le programme, encore partiellement schématique :

```

s := nil; u := m; v := n;
FAIRE
  [[résultat = f(s : u, v)]]
  SI u ≠ 0 ALORS s := s : (u-1)^v : (u-2) : ... : 1 : 0;
    u := 0; v := 1
  IS
  calculer s' et x tels que s = s' : 0^x
  v := v + x + 1
  SI s' = nil ALORS FINI IS
  u := der(s'); s := deb(s')
BOUCLER

```

4. REPRÉSENTATION DE LA SUITE

Pour développer complètement le programme, il faut se donner maintenant une représentation de s . Les seules opérations faites sur s sont :

- des concaténations à droite;
- l'extraction de termes à droite;
- un test pour savoir si la suite est vide.

Ceci confère à s la structure d'une pile. $\text{der}(s)$ est le sommet de la pile. Mais nous n'avons pour le moment aucun moyen de relier cette pile à celle qui est engendrée par l'exécution de la définition récursive sur un ordinateur.

Le programme contient une seule opération d'empilement :

$$s := s : (u-1)^v : (u-2) : \dots : 1 : 0.$$

La partie droite est ordonnée non croissante. Comme au début s est vide, et que cet empilement se produit lorsque le terme de droite de s vaut u , il est facile de montrer par récurrence que s est toujours une suite de termes ordonnée non croissante de gauche à droite. L'opération d'empilement est

faite au début avec $u = m$, et n'introduit que des entiers inférieurs à m . Compte tenu de tout ceci, on peut définir complètement s par un vecteur donnant le nombre d'occurrences de chaque entier de l'intervalle $0 : m - 1$ dans s . De manière précise, posons $c[i]$. nombre d'occurrences de la valeur $i - 1$ dans s (Remarque : ce choix est fait pour conserver les notations de notre article antérieur [2]. Il serait sans doute plus naturel de noter $c[i - 1]$ le nombre d'occurrences de $i - 1$ dans s).

Parce que s est toujours ordonnée non croissante, l'occurrence de $s : u$ dans l'invariant de boucle garantit que s ne contient que des entiers supérieurs ou égaux à u , et donc aucun entier inférieur à u , soit :

$$c[u] = c[u - 1] = \dots = c[1] = 0.$$

L'empilement $s : = s : (u - 1)^v : (u - 2) : \dots : 1 : 0$ est ainsi simplement réalisé par :

```
c[u] := v;
POUR i := 1 JUSQU'A u-1 FAIRE c[i] := 1 BOUCLER
```

Le nombre x de zéros à droite de s est donné par $c[1]$. Pour retirer ces zéros de s , il suffit de faire $c[1] := 0$.

Retirer le dernier élément de s pour le mettre en u est un peu plus difficile. Parce que s est ordonnée non croissante, son dernier élément est aussi son plus petit élément. Il faut donc chercher le plus petit nombre ayant une occurrence dans s , c'est-à-dire le plus petit i tel que $c[i + 1] \neq 0$. S'il n'y en a pas, c'est que s est vide :

```
u := 0;
FAIRE
  SI u = m ALORS FINI IS
  SI c[u + 1] ≠ 0 ALORS FINI IS
  u := u + 1
BOUCLER
```

Si, à la sortie de la boucle, $u = m$, c'est que s est vide. On peut alléger cette boucle en rajoutant une case au tableau c , soit $c[m + 1]$, que l'on initialise à 1. De la sorte, il y a toujours une case non nulle dans c , et la recherche se fait plus simplement par :

```
u := 0;
TANT QUE c[u + 1] = 0 FAIRE u := u + 1 BOUCLER
```

On obtient ainsi le programme définitif :

```

u : = m; v : = n; c[1 : m] : = 0; c[m + 1] : = 1
FAIRE
  SI u ≠ 0 ALORS c[u] : = v; v : = 1
    POUR i : = 1 JUSQU'A u - 1 FAIRE
      c[i] : = 1
    BOUCLER
  IS
  v : = v + c[1] + 1; c[1] : = 0
  u : = 0
  TANT QUE c[u + 1] = 0 FAIRE u : = u + 1 BOUCLER
  SI u = m ALORS FINI IS
  c[u + 1] : = c[u + 1] - 1
BOUCLER
résultat : = v

```

Ce même programme avait été obtenu [2] au prix de calculs beaucoup plus compliqués, en passant par un sous-programme récursif, que l'on ne pouvait transformer en programme itératif sans recourir à une pile. Nous avons démontré qu'elle était ordonnée non croissante, et donc représentable par le même vecteur c . Il en résulte que s est bien la pile engendrée par la conservation des arguments u lors des appels successifs de A .

Rappelons que cette forme utilise un seul vecteur de dimension n , et prend donc moins de place que la procédure de Rice [14] qui en demande deux. Ceci a déjà été discuté [2].

5. CONCLUSIONS

Un exemple ne suffit pas à établir un résultat de portée générale. Tout au plus peut-il suggérer quelque tendance. Le cas discuté ici paraît assez significatif. Nous avons publié une suite de calculs conduisant à une procédure itérative pour la fonction d'Ackermann, meilleure quant à l'encombrement en mémoire que la seule connue à l'époque, celle de Rice, que l'on croyait pourtant la meilleure possible. Il était déjà important de voir comment le calcul pouvait fournir des résultats que la seule imagination créatrice n'avait pu deviner. De tels exemples justifient ce que nous avons depuis appelé « la programmation analytique » par analogie avec la géométrie du même nom où le calcul se substitue à l'invention pour l'obtention du résultat.

Mais l'obstacle demeurait la complexité des calculs à faire pour obtenir le résultat, et, de ce point de vue, l'analogie avec la géométrie analytique tient toujours. De même que des mathématiciens comme Darboux ont réussi à simplifier considérablement les calculs de la géométrie analytique (Gaston Darboux, principes de géométrie analytique, Gauthier-Villars, Paris, 1917), il est assez vraisemblable que l'on arrivera à simplifier suffisamment les calculs

de la programmation analytique pour en faire un outil effectif en programmation.

L'exemple traité ici est une indication dans ce sens. Il renforce l'idée qu'une procédure récursive se prête mieux aux manipulations qu'une procédure itérative, et qu'il y a donc intérêt à faire le plus de chemin possible au niveau récursif, et ne passer à l'itératif qu'en fin de transformation. Le problème est que l'on manque pour le moment de repères pour la conduite des transformations de définitions récursives. On a quelques idées sur la façon de calculer la complexité d'une définition récursive [1, 8], mais c'est encore trop fragmentaire. Il y a là un important sujet de recherche.

BIBLIOGRAPHIE

1. J. ARSAC, *Les bases de la programmation*, Paris, Dunod, 1983.
2. J. ARSAC, *Emploi de méthodes constructives en programmation. Un dossier : la fonction d'Ackermann*, RAIRO, Inf. théor., vol. 11, n° 2, 1977, p. 91-112.
3. J. ARSAC et Yves KODRATOFF, *Some Techniques for Recursion Removal from Recursive Functions*, ACM Toplas, Vol. 4, n° 2, avril 1982, p. 295-322.
4. M. A. AUSLANDER et H. R. STRONG, *Systematic Recursion Removal*, Com A.C.M., vol. 21, n° 2, février 1978, p. 127-134.
5. F. L. BAUER et H. WOSNER, *Algorithmic Language and Program Development* Springer Verlag, Berlin, 1982.
6. R. M. BURSTALL et J. DARLINGTON, *A System which Automatically Improves Programs*, Acta informatica, vol. 6, 1976, p. 41-60.
7. R. M. BURSTALL et J. DARLINGTON, *A Transformation System for Developing Recursive Programs*, J.A.C.M., vol. 24, n° 1, janvier 1977, p. 44-67.
8. K. DJOSSOU, *Thèse de 3^e cycle*, université P.-et-M.-Curie, Paris, 1984.
9. G. HUET et B. LANG, *Proving and Applying Program Transformations Expressed with Second Order Patterns*, Acta informatica, vol. 11, n° 1, janvier 1978, p. 31-55.
10. L. KOTT, *About a Transformation System: Theoretical Study in Program Transformation*, in *Program Transformation: 3rd Symposium on Programming*, Dunod, Paris, 1978.
11. J. MCCARTHY, *Recursive Functions of Symbolic Expressions and their Computation by Machine*, Com. A.C.M., vpl. 16, n° 8, août 1972, p. 491-502.
12. Z. MANNA et R. WALDINGER, *Synthesis: Dreams → Programs*, Tech. Rep. CS 77-630, Computer sc. Dep., Stanford University, Californie, 1979.
13. P. PEPPER, H. PARTSCH, H. WOESSNER et F. L. BAUER, *Transformational Approach to Programming*, in *Program Transformation, 3rd Symposium on Programming*, Dunod, Paris, 1978, p. 248-262.
14. H. G. RICE, *Recursion and Iteration*, Com. A.C.M., vol. 8, 1965, p. 114-115.
15. J. S. ROHL, *Recursion via Pascal Sophisters and Calculators*, Australia, 1983.
16. B. WEGBREIT, *Goal Directed Program Transformation*, I.E.E.E. transactions in software engineering SE-2, 2, juin 1976, p. 69-80.