RAIRO. INFORMATIQUE THÉORIQUE

JACQUES J. ARSAC

Emploi de méthodes constructives en programmation. Un dossier : la fonction d'Ackermann

RAIRO. Informatique théorique, tome 11, n° 2 (1977), p. 91-112

<http://www.numdam.org/item?id=ITA_1977__11_2_91_0>

R.A I.R.O. Informatique théorique/Theoretical Computer Science
(vol. 11, n° 2, 1977, p 91 à 112)

EMPLOI DE MÉTHODES CONSTRUCTIVES EN PROGRAMMATION. UN DOSSIER : LA FONCTION D'ACKERMANN (1)

par Jacques J. ARSAC (2)

Communiqué par J -F PERROT

Résumé – *On construit une procédure itérative calculant la fonction d'Ackermann en se servant de méthodes générales deux façons de transformer une définition récursive en programme itératif, des transformations syntaxiques qui ne dépendent pas de la signification du programme, des transformations sémantiques locales qui n'en font intervenir que des propriétés superficielles, des transformations sémantiques profondes basées sur l'emploi d'assertions On suggère une façon de prouver le programme ainsi construit On discute la valeur de ces méthodes dans la pratique de la programmation*

1. INTRODUCTION

Nous nous proposons de montrer ici l'importance des transformations de programme comme outil de programmation, au service du praticien. Nous en utiliserons différents types, que nous essayons de classifier.

1.1. La transformation de procédures récursives en procédures itératives

La transformation de la définition récursive d'une fonction en une procédure itérative a été bien étudiée (voir par exemple Manna [18], Vuillemin [22]). Principalement en vue d'une automatisation, on opère habituellement en donnant des définitions récursives types, avec les schémas itératifs associés, et en identifiant une définition concrète avec un schéma abstrait (Burstall [7]). Il semble plus simple au programmeur de recourir à une méthode générale

(1) Reçu en octobre 1976 et en version définitive en janvier 1977.
(2) Institut de Programmation Université Pierre et Marie Curie, Paris.

(Arsac [4]), qui échappe aux limitations inhérentes à tout catalogue. On en trouvera un exemple ici.

La transformation d'une procédure récursive en une procédure itérative peut aussi être faite suivant la même technique d'identification de schémas types (Irlik [14]). Mais là aussi, l'emploi d'une méthode générale est plus souple quand aucun recours à l'automatisme n'est en jeu (Arsac [4]). On ne pourra la présenter dans le cadre restreint de ce dossier.

1.2. Les transformations syntaxiques

L'attention a été attirée sur ces transformations par l'affaire des branchements. Les transformations syntaxiques modifient la forme d'un programme en conservant l'histoire de ses calculs (Ledgard [17]). Elles ne modifient donc ni le temps de calcul, ni l'encombrement en mémoire. Mais elles affectent la lisibilité, et aussi la facilité de preuve. Elles ont été utilisées sous le nom de « node splitting techniques » dans les recherches sur l'élimination des instructions de branchement (Manna-Ashcroft [1], Floyd-Knuth [10], Peterson-Kasami [20], Kosaraju [16]). Certaines de ces transformations sont mentionnées comme « évidentes » à propos des manipulations à faire sur des programmes lors de leur création (Gerhardt [12], Knuth [15]). Leur liaison avec les structures de contrôle formées d'une itération avec instruction de sortie indicée (structure REPEAT-EXIT ou RE selon la nomenclature de Ledgard [17]) est indiquée par Nolin et Ruggiu [19]. Un catalogue des transformations élémentaires nécessaires aux besoins de la pratique a été proposé dès 1974 (Arsac [2]), et une construction formelle de ces transformations réalisée par G. Cousineau [9].

Une certaine suspicion leur reste attachée, et Ledgard [17] les déclare de peu de valeur pour la pratique. Nous y ferons très largement appel ici. Nous les justifierons par quelques considérations sémantiques, une preuve plus formelle étant hors des objectifs du présent dossier (*cf.* Arsac [4], Cousineau [9]).

1.3. Les transformations sémantiques locales

Alors que les transformations syntaxiques ne dépendent pas de la signification du programme, celles-ci reposent sur une connaissance limitée de son action, et, en tous cas, de la sémantique des instructions. Elles sont à peu près celles qu'effectue un compilateur optimiseur (Knuth [15]) : permutation de deux affectations à des variables indépendantes, fusion de deux affectations successives à une même variable, suppression d'un test lorsque la valeur du prédicat est connue...

1.4. Les transformations sémantiques profondes

Elles reposent sur des propriétés du programme étudié. On les fonde sur l'emploi des assertions inductives, proposées par Floyd [11], et formalisées

par Hoare [13]. Elles apparaissent au praticien comme une façon de suivre pas à pas l'exécution du programme, non dans des cas particuliers, mais de façon « abstraite », dans la généralité des cas. Loin d'être une contrainte supplémentaire pour le programmeur, elles interviennent plutôt comme une aide pour la création d'un programme (Arsac [4]).

Elles ont aussi un facteur essentiel de transformation. Ayant obtenu un programme, on en observe le comportement, soit par des exécutions « à la main », soit par des exécutions surveillées en machine, et l'on conjecture une propriété du programme. On la prouve, par exemple en utilisant la méthode des assertions. On répercute la propriété ainsi prouvée sur le programme pour le simplifier. Nous en donnerons un exemple.

1.5. Notations

Nous utiliserons ici les structures de contrôle d'EXEL (Vasseur [5]) qui sont aussi celles nommées REn par Ledgard [17] :

L'instruction de sélection sera écrite

SI ⟨ prédicat ⟩ ALORS ⟨ suite d'instructions ⟩ SINON
⟨ suite d'instructions ⟩ **IS**

l'emploi de la parenthèse fermante IS dispensant de parenthèses d'instructions autour de chaque alternant.

L'instruction d'itération sera enfermée entre des accolades formant parenthèses de boucle. La suite d'instructions entre accolades est exécutée répétitivement jusqu'à rencontre d'une instruction de sortie représentée par le signe « ! » (à prononcer EXIT), et qui signifie : sortir de l'itération englobante et passer en séquence. Il est généralisé en !p : sortir des p itérations immédiatement englobantes, et passer en séquence.

1.6. Le dossier présenté ici

Nous voulons illustrer la puissance des méthodes de construction de programme fondées sur l'emploi des outils que l'on vient d'énumérer. C'est pourquoi nous avons choisi d'étudier la fonction d'Ackermann, proposée comme le type d'une fonction définie récursivement et dont la forme itérative est difficile à obtenir. Cette fonction ayant été bien étudiée (Rice [21], Berry [8]), on a ainsi une référence à laquelle on pourra comparer les résultats obtenus. Nous attachons plus d'importance au chemin suivi pour atteindre le résultat, qu'à ce résultat lui-même. La méthode présentée ici est très générale, et a été employée avec succès sur de nombreux exemples, donnent en général de très bons programmes, et parfois des algorithmes nouveaux, comme c'est le cas pour la fonction d'Ackermann.

2. ÉLIMINATION D'UN NIVEAU DE RÉCURSIVITÉ

La fonction d'Ackermann est définie par

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

La fonction est appelée récursivement dans le cas où $n = 0$. Il faut essayer d'éviter celà pour obtenir une situation plus conventionnelle, où existe un cas où la fonction se calcule sans récursivité. Si l'on applique la définition récursive au cas $n = 0$, on trouve $A(m, 0) = A(m - 1, A(m, -1))$ qu'il faut identifier avec $A(m - 1,1)$. On est ainsi conduit très simplement à une nouvelle définition, où ne figurent plus que deux occurrences de A

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

Cette définition est utilisée par Rice dans son algorithme itératif, qui associe, pour toute valeur $m > 0$ du premier argument, la valeur 1 de la fonction à la valeur -1 du second argument. Il est impossible de dire si cette modification de la définition a joué un rôle essentiel dans la construction qui suit.

Nous utilisons la méthode générale suivante, toutes les fois qu'il s'agit de la définition récursive d'une fonction

> faire apparaître des suites récurrentes de valeurs des arguments sur lesquels portent les appels de la fonction.

On le fait très simplement en supposant que l'on a à calculer A pour les arguments $u(i - 1)$, $v(i - 1)$. Supposons qu'ils ne satisfassent pas les conditions où il n'y a pas récursivité, c'est-à-dire $u(i - 1) \neq 0 \wedge v(i - 1) \geqslant 0$

$$A(u(i - 1), v(i - 1)) = A(u(i - 1) - 1, A(u(i - 1), v(i - 1) - 1))$$

Nous sommes ainsi amenés à appliquer A à une nouvelle paire d'arguments, que nous nommons $u(i)$ et $v(i)$. En identifiant

1) $u(i) = u(i - 1) - 1$
2) $v(i) = A(u(i - 1), v(i - 1) - 1)$

Grâce à cette définition, nous avons

$$A(u(i - 1), v(i - 1)) = A(u(i), v(i))$$

de sorte que cette quantité est invariante sur i. Prenons

3) $u(0) = m$

4) $v(0) = n$

$$A(u(i), v(i)) = A(u(0), v(0)) = A(m, n)$$

est le résultat cherché.

Désignons par k le plus petit i tel que

$$u(i) = 0 \quad \text{OU} \quad v(i) < 0$$

Notons ceci

5) $k = \text{MIN } i : u(i) = 0 \text{ OU } v(i) < 0$

Son existence est garantie par le fait que la suite $u(i)$ est strictement décroissante. Alors, par la définition de la fonction

6) $A(u(k), v(k)) = \text{SI } u(k) = 0 \text{ ALORS } v(k) + 1 \text{ SINON } 1$

L'ensemble des relations 1) à 6) définit un algorithme calculant la fonction d'Ackermann, suivant les notations des *Nouvelles leçons de programmation* [4]. Il est très facile à retranscrire dans les notations LUCID de Ashcroft [6]

next $u = u - 1$

next $v = A(u, v - 1)$

first $u = m$

first $v = n$

$r = \text{SI } u = 0 \text{ ALORS } v + 1 \text{ SINON } 1$ *as soon as* $u = 0 \text{ OU } v < 0$

On tire aisément un programme de cet algorithme

P1
```
u ← m; v ← n;
TANT QUE u ≠ 0 ET v ≥ 0 FAIRE v ← A(u, v − 1); u ← u − 1 FTQ;
        r ← SI u = 0 ALORS v + 1 SINON 1
```

Il est formé d'une boucle, la fonction A étant appelée récursivement dans la boucle. Mais il n'y a plus qu'une seule occurrence de A. Ne cherchons pas à le discuter. Mais utilisons une nouvelle méthode générale pour éliminer la récursivité.

3. ÉLIMINATION DU DEUXIÈME NIVEAU DE RÉCURSIVITÉ

Nous utilisons la méthode générale suivante :

> En transformant d'abord les appels récursifs pour qu'ils portent toujours sur les mêmes paramètres effectifs, remplacer la procédure récursive avec paramètres par une procédure récursive sans paramètres, sans variables locales.

L'appel récursif de A dans la boucle est dans l'instruction

$$v \leftarrow A(u, v - 1)$$

Nous utiliserons une transformation sémantique fréquente dans les compilateurs optimiseurs. La séquence de 2 affectations à une même variable x

$$x \leftarrow g1 \quad ; \quad x \leftarrow g2(x, t)$$

où t désigne une quelconque liste de variables distinctes de x, peut être remplacée par l'affectation unique

$$x \leftarrow g2(g1, t)$$

et réciproquement. On peut ainsi décomposer l'affectation $v \leftarrow A(u, v - 1)$ en la séquence

$$v \leftarrow v - 1 \quad ; \quad v \leftarrow A(u, v)$$

Nommons $\mathrm{AP}(u, v)$ la procédure exécutant l'action $v \leftarrow A(u, v)$. Le programme calculant $A(m, n)$ et donnant le résultat dans v s'écrit alors :

$$u \leftarrow m \quad ; \quad v \leftarrow n \quad ; \quad \mathrm{AP}(u, v)$$

Quant à $\mathrm{AP}(u, v)$, on peut le définir par

P2

```
PROCEDURE AP(u, v); DEBUT
    TANT QUE u ≠ 0 ET v ⩾ 0 FAIRE v ← v − 1; AP(u, v); u ← u − 1 FTQ;
        v ← SI u = 0 ALORS v + 1 SINON 1 FIN
```

Il faut maintenant transformer cette procédure en une autre, encore récursive, mais prenant u et v comme variables globales. L'exécution de cette nouvelle procédure altère les valeurs de u et v. Comme v est une variable résultat de AP, sa redéfinition dans la procédure ne pose aucun problème. Par contre l'altération de u rend impossible son utilisation après l'appel de AP. Il faut donc préserver u dans une pile avant l'appel récursif, et restituer sa valeur en la reprenant dans la pile après cet appel. Enfin la pile doit être initialisée avant le premier appel. D'où le programme

$$u \leftarrow m \quad ; \quad v \leftarrow n \quad ; \quad \text{initialiser la pile} \quad ; \quad \mathrm{AQ}$$

et la procédure

P3

```
PROCEDURE AQ; DEBUT
    TANT QUE u ≠ 0 ET v ⩾ 0 FAIRE v ← v − 1; emp(u); AQ;
        dep(u); u ← u − 1 FTQ;
        v ← SI u = 0 ALORS v + 1 SINON 1
            FIN
```

La transformation de cette procédure récursive sans paramètres formels ni variables locales se fait par recours à des équations de programme (Arsac 4, Cousineau 9). Nous donnons le résultat sans calculs intermédiaires, demandant au lecteur d'en vérifier la validité.

P4

```
u ← m; v ← n; initialiser la pile;
{{ SI u ≠ 0 ET v ⩾ 0 ALORS v ← v − 1; emp(u) SINON ! IS };
   v ← SI u = 0 ALORS v + 1 SINON 1;
   SI pile vide ALORS ! SINON dep(u); u ← u − 1 IS }
```

4. STRUCTURE DE LA PILE

Pour simplifier la discussion, nous supposons dorénavant $m > 0$ ET $n \geqslant 0$ (s'il n'en est pas ainsi, le résultat se calcule immédiatement). A l'entrée dans la grande boucle, la pile est vide, $u > 0$ et $v \geqslant 0$. On va donc répéter l'action d'empiler u et décrémenter v jusqu'à rendre celui-ci négatif. On aura dans la pile un certain nombre d'éléments égaux à m. A la sortie de la boucle intérieure, v reprendra la valeur 1, puis, la pile étant non vide, u sera décrémenté, prenant la valeur $m - 1$, qui sera à son tour empilée peut-être plusieurs fois, et ainsi de suite. On peut donc conjecturer que la pile va contenir une suite non croissante d'entiers entre 1 et m. Nous laissons au lecteur le soin de prouver le théorème.

THÉORÈME : *si la pile est non vide, alors, en l'ordonnant de la base vers le sommet, elle contient une suite de valeurs entières non croissantes, la valeur de la base étant inférieure ou égale à m, la valeur du sommet étant strictement positive.*

Une façon simple de le prouver consiste à prendre cet énoncé comme assertion en tête de la grande boucle de P4, puis de montrer qu'elle est invariante sur chacune des 2 boucles, et vraie par l'initialisation.

5. AUTRE REPRÉSENTATION DE LA PILE

Nous avons maintenant une information très forte sur la structure de la pile. Nous savons qu'elle est faite d'entiers (la pile est constituée par emp(u), et u est entier) pris dans l'intervalle $[1 : m]$, et qu'elle est non croissante. Elle a donc nécessairement la forme

$$p_1^{q_1} p_2^{q_2} p_3^{q_3} \ldots p_k^{q_k}$$

où $p_i^{q_i}$ représente la concaténation de q_i valeurs égales à p_i, avec

$$m \geqslant p_1 > p_2 > p_3 > \ldots > p_k > 0$$

Donnons-nous un vecteur $c[1:m]$ tel que :

$c[j] = 0$ si j n'a pas d'occurrence dans la pile

$c[j] = q_i$ si $j = p_i$ est un élément de la pile.

On reconstitue la pile à partir du vecteur $c[1:m]$ en empilant $c[j]$ fois la valeur j, pour j décroissant de m à 1. Ainsi la connaissance du vecteur $c[1:m]$ est équivalente à celle de la pile. On est ainsi assuré de pouvoir construire une procédure itérative en utilisant les variables u, v et le seul vecteur $c[1:m]$. Mais ne nous arrêtons pas en si bon chemin, et voyons comment elle est faite. Nous avons travaillé en « programmation descendante » en prenant bien garde à ne rien dire de la représentation de la pile, nous contentant d'affirmer l'existence des opérations :

initialiser la pile
emp (u)
dep (u)
pile vide

Si nous adoptons la représentation de la pile par le vecteur $c[1:m]$

initialiser la pile, c'est faire $c[j] = 0$ pour tout j $1 \leqslant j \leqslant m$

emp (u). 2 cas sont possibles.

Si u est déjà dans la pile, emp (u) accroît de 1 le nombre de ses occurrences, et se réduit à

$$c[u] \leftarrow c[u] + 1$$

Si u n'est pas dans la pile, alors $c[u] = 0$. Pour l'empiler, il faut faire $c[u] = 1$. On y arrive encore par $c[u] \leftarrow c[u] + 1$, qui réalise emp (u) dans tous les cas.

Dep (u) nécessite de trouver la valeur du sommet, puis d'en réduire de 1 unité le compte d'occurrences. Le sommet est le plus petit élément dans la pile, puisqu'elle est non croissante. C'est donc le plus petit u tel que $c[u] \neq 0$.

dep (u) : trouver le plus petit u tel que $c[u] \neq 0$; $c[u] \leftarrow c[u] - 1$

Pile vide la pile est vide si et seulement si elle ne contient aucun élément, c'est-à-dire si $c[j] = 0$ pour tout j, $1 \leqslant j \leqslant m$.

Nous obtenons ainsi le programme

P5

```
u ← m; v ← n; faire c[1:m] = 0;
{{ SI u ≠ 0 ET v ⩾ 0 ALORS v ← v − 1; c[u] ← c[u] + 1 SINON ! IS };
    v ← SI u = 0 ALORS v + 1 SINON 1;
  SI les c[1:m] sont nuls ALORS !
        SINON trouver le plus petit u tel que c[u] ≠ 0;
              c[u] ← c[u] − 1; u ← u − 1 IS }
```

On pourrait garder ce programme, mais il est facile de l'améliorer. Nous devons en effet successivement regarder si tous les $c[1 : m]$ sont nuls, et sinon trouver le plus petit non nul. Ces deux opérations n'en font qu'une. Nous obtenons ainsi un programme équivalent un peu plus simple

P6

```
u ← m; v ← n; faire c[1:m] = 0;
{{ SI u ≠ 0 ET v ≥ 0 ALORS v ← v − 1; c[u] ← c[u] + 1 SINON! IS };
      v ← SI u = 0 ALORS v + 1 SINON 1;
      chercher le plus petit u tel que c[u] ≠ 0;
   SI existe pas ALORS!
                SINON c[u] ← c[u] − 1; u ← u − 1 IS }
```

6. AMÉLIORATIONS

Considérons d'abord la boucle intérieure

$\{$ SI $u \geqslant 0$ ET $v \geqslant 0$ ALORS $v \leftarrow v - 1$; $c[u] \leftarrow c[u] + 1$ SINON ! IS $\}$

Les affectations $v \leftarrow v - 1$; $c[u] \leftarrow c[u] + 1$ laissent la somme $c[u] + v$ invariante (u est une constante sur la boucle), de sorte que

$$c[u]_{\text{final}} + v_{\text{final}} = c[u]_{\text{initial}} + v_{\text{initial}}$$

Si à l'entrée de la boucle, $u = 0$, rien n'est fait. Sinon, la boucle est effectuée, et la valeur finale de v est -1. Ainsi, dans ce cas

$$c[u]_{\text{final}} = c[u]_{\text{initial}} + v_{\text{initial}} + 1$$

On peut donc remplacer la boucle par

SI $u = 0$ ALORS SINON $c[u] \leftarrow c[u] + v + 1$; $v \leftarrow -1$ IS

L'affectation $v \leftarrow$ SI $u = 0$ ALORS $v + 1$ SINON 1 peut être changée en

SI $u = 0$ ALORS $v \leftarrow v + 1$ SINON $v \leftarrow 1$ IS

Le début de la grande boucle du programme devient ainsi

SI $u = 0$ ALORS SINON $c[u] \leftarrow c[u] + v + 1$; $v \leftarrow -1$ IS;
SI $u = 0$ ALORS $v \leftarrow v + 1$ SINON $v \leftarrow 1$ IS

Utilisons l'absorption (NR1, GE1, AR3)

SI θ ALORS α SINON β IS ; γ est équivalent à
SI θ ALORS α ; γ SINON β ; γ IS

On fait ainsi entrer la deuxième sélection SI $u = 0$ dans les alternants de la première

SI $u = 0$ ALORS SI $u = 0$ ALORS $v \leftarrow v + 1$ SINON $v \leftarrow 1$ IS
SINON $c[u] \leftarrow c[u] + v + 1$;
$v \leftarrow -1$; SI $u = 0$ ALORS $v \leftarrow v + 1$ SINON $v \leftarrow 1$ IS
IS

Mais cette fois, la valeur du prédicat $u = 0$ dans les sélections intérieures est connue Supprimant les tests inutiles et les alternants qui ne peuvent être atteints, et changeant la séquence $v \leftarrow -1$; $v \leftarrow 1$ en $v \leftarrow 1$, il reste

SI $u = 0$ ALORS $v \leftarrow v + 1$
SINON $c[u] \leftarrow c[u] + v + 1$; $v \leftarrow 1$ IS

Continuons l'absorption en faisant entrer dans cette sélection la fin de la boucle Elle devient

{ SI $u = 0$ ALORS $v \leftarrow v + 1$; chercher le plus petit u tel que $c[u] \neq 0$;
SI existe pas ALORS ! SINON $c[u] \leftarrow c[u] - 1$;
$u \leftarrow u - 1$ IS
SINON $c[u] \leftarrow c[u] + v + 1$, $v \leftarrow 1$;
chercher le plus petit u tel que $c[u] \geqslant 0$;
SI existe pas ALORS ! SINON $c[u] \leftarrow c[u] - 1$;
$u \leftarrow u - 1$ IS
IS }

Nous avons souligné que l'opération emp (u) porte toujours sur le plus petit élément présent dans la pile, de sorte que après avoir fait $c[u] \leftarrow c[u] + v + 1$, la reponse a « chercher le plus petit u tel que $c[u] \neq 0$ » est formee de la même valeur de u Ne modifiant pas u, c'est une action vide Elle implique qu'il existe un u tel que $c[u] \neq 0$, et donc le prédicat « existe pas » a la valeur FAUX Supprimant l'action vide et le prédicat inutile, il reste

{ SI $u = 0$ ALORS $v \leftarrow v + 1$, chercher le plus petit u tel que $c[u] \neq 0$;
SI existe pas ALORS ! SINON $c[u] \leftarrow c[u] - 1$;
$u \leftarrow u - 1$ IS
SINON $c[u] \leftarrow c[u] + v + 1$; $v \leftarrow 1$;
$c[u] \leftarrow c[u] - 1$; $u \leftarrow u - 1$ IS }

Suivant le rappel fait au paragraphe 3, la séquence

$$c[u] \leftarrow c[u] + v + 1 \quad ; \quad v \leftarrow 1 \quad ; \quad c[u] \leftarrow c[u] - 1$$

est condensée en $c[u] \leftarrow c[u] + v$; $v \leftarrow 1$

Pour alléger l'écriture, abrégeons

chercher le plus petit u tel que $c[u] \neq 0$ en u min

existe pas en $\bar{e}$

Nous avons maintenant le programme (aux initialisations près)

P7

```
{ SI u = 0 ALORS v ← v + 1 ; u min ;
        SI ē ALORS ! SINON c[u] ← c[u] − 1 ; u ← u − 1 IS
        SINON c[u] ← c[u] + v ; v ← 1 ; u ← u − 1 IS }
```

7.2. Transformations syntaxiques simples

Nous donnons ici, sans justification, 2 transformations syntaxiques dont nous avons besoin. Elles sont des cas particuliers de transformations plus générales décrites ailleurs (2, 9). Au lecteur désireux d'en percevoir la signification, nous conseillons de tracer les organigrammes des formes proposées comme équivalentes. Mais la véritable justification est ailleurs que dans une constatation sur un dessin.

On désigne par α, β, γ des suites d'instructions à sortie séquentielle (affectations, boucles TANT QUE ...)

φ une suite d'instructions quelconque

θ un prédicat.

L'équivalence entre suites d'instructions est notée $\doteqdot$

T1 $\{ \alpha ; \varphi \} \doteqdot \alpha \quad ; \quad \{ \varphi ; \alpha \}$

T2 $\{ \alpha ;$ SI θ ALORS φ SINON β IS $\} \doteqdot \{ \{ \alpha ;$ SI θ ALORS ! SINON β IS $\} ; \varphi \}$

Nous allons nous en servir pour modifier le programme déjà pas mal transformé.

Nous utilisons d'abord l'heuristique suivante, excellente toutes les fois que l'on n'a pas d'objectif précis :

en utilisant T2, s'arranger pour que chaque boucle n'ait qu'un seul point de remontée (point qui renvoie en tête de boucle et provoque l'itération).

Dans le programme que nous étudions il y a deux points de remontée :

l'un après $u \leftarrow u - 1$ dans l'alternant FAUX du premier alternant de la sélection

SI $u = 0$ ...

l'autre après $u \leftarrow u - 1$, dans l'alternant FAUX de cette même sélection (on constate ainsi qu'il y a une même affectation avant chaque remontée. On

aurait pu la « mettre en facteur » à droite par réciproque de l'absorption. En fait, c'est une impasse qui ne mène nulle part). Appliquons T2 :

$$\{\{\text{SI } u = 0 \text{ ALORS ! SINON } c[u] \leftarrow c[u] + v;\ v \leftarrow 1;\ u \leftarrow u - 1 \text{ IS}\};\ v \leftarrow v + 1;\ u \text{ min};\ \text{SI } \bar{e} \text{ ALORS ! SINON } c[u] \leftarrow c[u] - 1;\ u \leftarrow u - 1 \text{ IS}\}$$

Examinons maintenant ce qui se passe quand, après la boucle intérieure, $c[1] \neq 0$. L'opération u min qui donne le plus petit u tel que $c[u] \neq 0$ donne alors $u = 1$. $\bar{e}$ est FAUX, et donc $c[1]$ est décrémenté de 1, et l'on fait

$$u \leftarrow u - 1, \qquad \text{donc} \quad u = 0.$$

On revient en tête de la grande boucle avec $u = 0$, et la boucle intérieure n'est pas effectuée. Le processus recommence tant que $c[1] \neq 0$. Il y a donc une boucle implicite commandée par le test $c[1] \neq 0$.

Nous prenons comme objectif des transformations de la faire apparaître. Nous avons constaté que la séquence

$$u \text{ min} \quad ; \quad \ldots . \text{ IS}$$

avait un comportement spécial quand $c[1] \neq 0$. En utilisant le fait que pour tout φ $\varphi \doteq$ SI θ ALORS φ SINON φ IS, nous enfermons la séquence u min ... IS dans un test (l'existence de $c[1]$ est garantie par $m > 0$)

$$\text{SI } c[1] \neq 0 \text{ ALORS } u \text{ min} \quad ; \quad \text{SI } \bar{e} \text{ ALORS ! SINON } c[u] \leftarrow c[u] - 1;\ u \leftarrow u - 1 \text{ IS}$$
$$\text{SINON } u \text{ min} \quad ; \quad \text{SI } \bar{e} \text{ ALORS ! SINON } c[u] \leftarrow c[u] - 1;\ u \leftarrow u - 1 \text{ IS}$$
$$\text{IS}$$

Dans l'alternant VRAI, $c[1] \neq 0 \Rightarrow u$ min $\rightarrow u = 1$, et donc $\bar{e}$ est FAUX. Dans l'alternant FAUX, u min $\rightarrow u > 1$, et donc à la fin $u > 0$. Compte tenu de ceci

$$\text{SI } c[1] \neq 0 \text{ ALORS } c[1] \leftarrow c[1] - 1 \quad ; \quad u \leftarrow 0$$
$$\text{SINON } u \text{ min} \quad ; \quad \text{SI } \bar{e} \text{ ALORS ! SINON } c[u] \leftarrow c[u] - 1;\ u \leftarrow u - 1$$
$$\text{« } u > 0 \text{ » IS} \quad \text{IS}$$

Le pas suivant de la grande boucle va se faire avec des hypothèses très différentes suivant l'alternant du test sur $c[1]$. Appelons β la boucle intérieure

$$\beta = \{\text{ SI } u = 0 \text{ ALORS ! SINON } c[u] \leftarrow c[u] + v;\ v \leftarrow 1;\ u \leftarrow u - 1 \text{ IS }\}.$$

Nous notons qu'après β, $u = 0$. Enfin β est à sortie séquentielle (c'est la boucle TANT QUE $u \neq 0$ FAIRE $c[u] \leftarrow c[u] + v;\ v \leftarrow 1;\ u \leftarrow u - 1$ FTQ).

Notre programme est devenu

$\{\beta; v \leftarrow v + 1; \langle\langle u = 0 \rangle\rangle$
SI $c[1] \neq 0$ ALORS $c[1] \leftarrow c[1] - 1; u \leftarrow 0$
SINON u min; SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1;$
$u \leftarrow u - 1$ IS
IS $\}$

Pour faire apparaître l'enchaînement de la sélection SI $c[1] \neq 0$ avec β, nous utilisons T1

β ; $\{v \leftarrow v + 1; \langle\langle u = 0 \rangle\rangle$
SI $c[1] \neq 0$ ALORS $c[1] \leftarrow c[1] - 1$; $u \leftarrow 0$
SINON u min;
SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1; u \leftarrow u - 1$ IS
IS; β $\}$

Puis nous absorbons β dans la sélection, en tenant pour évident que $!; \beta \equiv !$ Ceci fait apparaître la séquence

$$\langle\langle u = 0 \rangle\rangle \; c[1] \leftarrow c[1] - 1 \quad ; \quad u \leftarrow 0 \quad ; \quad \beta$$

Or sous l'hypothèse $u = 0$, $u \leftarrow 0$ ne modifie pas u, et β n'est pas effectué. On supprime donc ces instructions sans effet. Il reste

β ; $\{v \leftarrow v + 1$; SI $c[1] \neq 0$ ALORS $c[1] \leftarrow c[1] - 1$
SINON u min; SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1;$
$u \leftarrow u - 1; \beta$ IS
IS $\}$

Il y a de nouveau deux points de remontée. Suivant les conseils de l'heuristique déjà mentionnée, on utilise T2

P8 $\beta; \{\{v \leftarrow v + 1$; SI $c[1] \neq 0$ ALORS $c[1] \leftarrow c[1] - 1$ SINON ! IS $\}$;
u min; SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1$; $u \leftarrow u - 1$; β IS $\}$

Nous avons mis en évidence la boucle commandée par $c[1] \neq 0$. Pour avoir le test en tête de boucle, nous utilisons encore une fois T1, et cette boucle devient

$v \leftarrow v + 1$; $\{$ SI $c[1] \neq 0$ ALORS $c[1] \leftarrow c[1] - 1;$
$v \leftarrow v + 1$ SINON ! IS $\}$

On constate qu'elle laisse la somme $c[1] + v$ inchangée, et qu'à la sortie $c[1] = 0$. Ainsi son effet peut être caractérisé par

$$c[1]_{\text{final}} + v_{\text{final}} = c[1]_{\text{initial}} + v_{\text{initial}} \text{ ET } c[1]_{\text{final}} = 0$$

On peut donc la remplacer par

$$v \leftarrow v + c[1] \quad ; \quad c[1] \leftarrow 0$$

D'où le programme

β ; $\{ v \leftarrow v + 1; v \leftarrow v + c[1]; c[1] \leftarrow 0;$
u min ; SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1; u \leftarrow u - 1;$ β IS $\}$

La séquence $v \leftarrow v + 1; v \leftarrow v + c[1]$ est condensée en $v \leftarrow v + c[1] + 1$. Par réciproque de l'absorption, β est placée hors de la sélection comme dernière instruction de la boucle, puis ramené en tête de boucle par réciproque de T1. Enfin sa valeur est rétablie

P9
$\{\{$ SI $u = 0$ ALORS ! SINON $c[u] \leftarrow c[u] + v;$ $v \leftarrow 1;$ $u \leftarrow u - 1$ IS $\};$
$v \leftarrow v + c[1] + 1; c[1] \leftarrow 0;$ u min ;
SI $\bar{e}$ ALORS ! SINON $c[u] \leftarrow c[u] - 1; u \leftarrow u - 1$ IS $\}$

Il reste à détailler u min et à rétablir les initialisations pour avoir la procédure calculant $v \quad A(m, n)$

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

La transformation qui suit pourrait être faite sur des bases plus syntaxiques. Nous la prenons pour évidente, afin d'économiser l'appareil nécessaire. Quand $u > m$ dans la boucle intérieure, on sort de celle-ci, puis aussitôt du programme. Autrement dit, $u > m$ fait sortir des 2 boucles à la fois. On note ceci !2, dans la boucle intérieure. Moyennant quoi, si l'on en sort en séquence, c'est que $u \leqslant m$ et la sélection qui suit est inutile. On la supprime.

Il vient ainsi P10

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

8. INTERPRÉTATION

Ce programme a été « fabriqué », au sens littéral du terme, non à partir d'une stratégie, mais suivant des méthodes générales du travail. Il est valide parce que déduit d'une forme de départ valide (la forme récursive, qui définit la fonction) par des transformations valides. Les transformations sémantiques profondes ont été fondées sur des propriétés prouvées. Il n'est pas besoin d'une nouvelle preuve pour ce programme. Mais il faut lui donner un sens, en découvrant sa stratégie. Pour ce faire, il faut en améliorer la lisibilité, en rendant plus claires ses actions.

Comme cela s'est déjà produit, cet objectif est trop lointain pour guider effectivement les transformations à faire. Nous allons encore une fois examiner avec soin ce que fait le programme, et faire apparaître distinctement chacune de ses actions.

Nous avons supposé qu'il n'était mis en œuvre que pour $m > 0$ et $n \geqslant 0$. Dès lors, à la première entrée dans la boucle intérieure, $u > 0$. Nous savons qu'il en est encore ainsi à la fin de chaque pas de la grande boucle (u min et NON $\bar{e} \Rightarrow$ il existe $u \leqslant m$ tel que $c[u] \neq 0$. Comme $c[1] = 0$, $1 < u$. Après $u \leftarrow u - 1$, $u > 0$).

Nous avons donc l'assertion invariante $u > 0$ en tête du corps de la grande boucle :

$$\text{« } u > 0 \text{ » } \{ \text{ SI } u = 0 \text{ ALORS ! SINON } c[u] \leftarrow c[u] + v; \quad v \leftarrow 1; u \leftarrow u - 1 \text{ IS } \}$$

Utilisons ici des considérations sémantiques évidentes, pour ne pas prendre un chemin trop long de transformations syntaxiques et sémantiques locales. Sous l'hypothèse $u \neq 0$, le premier pas de la boucle est toujours exécuté. On le sort donc de la boucle, sans le test inutile sur u

$$c[u] \leftarrow c[u] + v \quad ; \quad v \leftarrow 1 \quad ; \quad u \leftarrow u - 1;$$
$$\{ \text{ SI } u = 0 \text{ ALORS ! SINON } c[u] \leftarrow c[u] + v; v \leftarrow 1; u \leftarrow u - 1 \text{ IS } \}$$

Maintenant $v = 1$ est un invariant sur la boucle. On peut donc remplacer v par 1 dans celle-ci, et en retirer l'affectation $v \leftarrow 1$, sans effet. Enfin, on a avant cette boucle l'assertion $c[1 : u] = 0$, vraie : la première fois à cause de l'initialisation $c[1 : m] \leftarrow 0$, aux pas suivants à cause de u min suivi de $u \leftarrow u - 1$. On remplace donc les affectations

$$c[u] \leftarrow c[u] + v \quad \text{par} \quad c[u] \leftarrow v.$$

Il reste

$$c[u] \leftarrow v \;\; ; \;\; u \leftarrow u - 1 ;$$
$$\{ \text{SI } u = 0 \text{ ALORS! SINON } c[u] \leftarrow 1 ; u \leftarrow u - 1 \text{ IS} \}$$

P10 a la forme

$$\alpha ; \{ c[u] \leftarrow v ; u \leftarrow u - 1 ; \varphi \}$$

Utilisant T1

$$\alpha ; c[u] \leftarrow v ; u \leftarrow u - 1 ; \{ \varphi ; c[u] \leftarrow u ; u \leftarrow u - 1 \}$$

Détaillons, en nous rappelant qu'après la boucle intérieure $v = 1$

$$\{ \{ \text{SI } u = 0 \text{ ALORS! SINON } c[u] \leftarrow 1 ; u \leftarrow u - 1 \text{ IS} \} ;$$
$$v \leftarrow c[1] + 2 ; c[1] \leftarrow 0 ; u \min ;$$
$$\text{SI } \bar{e} \text{ ALORS! SINON IS} ;$$
$$c[u] \leftarrow c[u] - 1 ; u \leftarrow u - 1 ; c[u] \leftarrow v ; u \leftarrow u - 1 \}$$

Nous avons repris une forme abrégée de l'action u min et du test $\bar{e}$ pour simplifier les manipulations du programme. Nous allons avoir à discuter l'évolution du vecteur c, et il ne paraît pas opportun que soient groupées en une boucle plusieurs opérations le concernant. Utilisant la réciproque de T2, on obtient P11. On simplifie l'initialisation, la mise à 0 de $c[1 : m]$ avant la boucle étant inutile.

P11
$m > 0$

```
        c[m] ← n ; u ← m − 1 ;
{ « A » SI u = 0
        ALORS « D » v ← c[1] + 2 ; c[1] ← 0 ;
              « B » u min ; SI ē ALORS! SINON IS ;
              « C » c[u] ← c[u] − 1 ; u ← u − 1 ; c[u] ← v ; u ← u − 1
        SINON c[u] ← 1 ; u ← u − 1
      IS }
```

Nous ne faisons pas ici une preuve, nous cherchons comment la faire. Nous opérons en omettant la discussion des cas particuliers, elle viendra plus tard. Après l'initialisation, nous avons

$$A : \; c[m] = n \text{ ET } c[1 : m - 1] = 0 \text{ ET } u = m - 1$$

Si le programme est correct, après un certain nombre de pas de la boucle, le programme doit s'arrêter avec $v = A(m, n)$. S'il s'arrête, c'est que l'on a $\bar{e}$, et donc que le vecteur $c[1 : m] = 0$. Donc

$$A \xrightarrow{*} B : \; c[1 : m] = 0 \text{ ET } v = A(m, n)$$

Comme nous savons que c est une représentation d'une pile, il est normal de supposer que le comportement du programme n'est pas influencé par ce que le vecteur c pourraît contenir au delà de m. Nous faisons dont l'hypothèse

H : l'assertion $A : c[1:j-1] = 0$ ET $c[j] = q$ ET $u = j - 1$

$$\xrightarrow{*} B : c[1:j] = 0 \text{ ET } v = A(j, q)$$
$$\forall\ j : 2 \leqslant j \leqslant p - 1$$

($\xrightarrow{*}$ voulant dire : donne après un certain nombre de pas de la boucle).

Regardons alors ce qui se passe pour

$A : \quad c[1:p-1] = 0$ ET $c[p] = q$ ET $u = p - 1 > 0$

Comme $u \neq 0$, on éxécute $c[u] \leftarrow 1$; $u \leftarrow u - 1$ et l'on se retrouve en A avec $A : c[p] = q$ ET $c[p-1] = 1$ ET $c[1:p-2] = 0$ ET $u = p - 2$.

Par H, on arrive à B :

$$c[p] = q \text{ ET } c[1:p-1] = 0 \text{ ET } v = A(p-1, 1)$$

Alors u min donne $u = p$,

$C : \quad c[p] = q$ ET $c[1:p-1] = 0$ ET $v = A(p-1, 1)$ ET $u = p$

Exécutant la séquence après C, on arrive en A avec

$A : \quad c[p] = q - 1$ ET $c[p-1] = A(p-1, 1)$ ET $c[1:p-2] = 0$
ET $u = p - 2$

Mais par les propriétés de la fonction d'Ackermann $A(p-1, 1) = A(p, 0)$

$A : \quad c[p] = q - 1$ ET $c[p-1] = A(p, 0)$ ET $c[1:p-2] = 0$
ET $u = p - 2$

Par H, $A \xrightarrow{*} B$

$B : \quad c[p] = q - 1$ ET $c[1:p-1] = 0$ ET $v = A(p-1, A(p, 0)) = A(p, 1)$

Comme ci-dessus, u min donne $u = p$, et l'on se retrouve en A avec

$A : \quad c[p] = q - 2$ ET $c[p-1] = A(p, 1)$ ET $c[1:p-2] = 0$ ET
$u = p - 2$

Généralisons par récurrence. Supposons atteint

$A : \quad c[p] = q - k$ ET $c[p-1] = A(p, k-1)$ ET $c[1:p-2] = 0$
ET $u = p - 2$

Par l'hypothèse H, $A \xrightarrow{*} B$

$B : \quad c[p] = q - k$ ET $c[1:p-1] = 0$
ET $v = A(p-1, A(p, k-1)) = A(p, k)$

qui donne :

$$A : \; c[p] = q - k - 1 \text{ ET } c[p-1] = A(p,k) \text{ ET } c[1:p-2] = 0 \text{ ET } u = p - 2$$

qui est le même état, après changement de k en $k+1$. Appliquant ceci avec $k = q$, on arrivera à

$$A : \; c[p] = 0 \text{ ET } c[p-1] = A(p, q-1) \text{ ET } c[1:p-2] = 0 \text{ ET } u = p - 2$$
$$\overset{*}{\to} B : \; c[p] = 0 \text{ ET } c[1:p-1] = 0 \text{ ET } v = A(p-1, A(p, q-1)) \Rightarrow$$
$$c[1:p] = 0 \text{ ET } v = A(p,q)$$

On peut ainsi prouver H par récurrence : si H est vraie jusqu'à $p-1$, c'est vrai pour p. Notre objectif n'est pas de poursuivre une preuve sur cette base. Tout ceci n'a été fait que pour suggérer un invariant possible. La valeur q qui peut se trouver en $c[p]$ quand $c[1:p-1] = 0$ est de la forme $A(p+1, k'-1)$. En général on aura $c[p] = A(p+1, k'-1) - k$. On est ainsi conduit à introduire un vecteur $k[1:m]$

$$D : \; c[m] = n - k[m]$$
$$c[j] = A(j+1, k[j+1] - 1) - k[j] \quad \forall j : 1 \leqslant j \leqslant m-1$$
$$k[1] = 0 \quad k[i] \geqslant 0 \quad \forall i : 2 \leqslant i \leqslant m$$

On prouve la correction du programme en montrant que D est invariant sur la boucle, et réalisé à partir de l'initialisation. Pour l'essentiel, supposant D vrai, on atteint C avec

$$D \text{ ET } c[1:u-1] = 0 \text{ ET } c[u] \neq 0 \text{ ET } v = c[1] + 2$$

Toujours en ne considérant que le cas général,

$$v = c[1] + 2 = A(2, k[2] - 1) + 2 = A(1, A(2, k[2] - 1)) = A(2, k[2])$$

en tenant compte des propriétés de A, et de $A(1,n) = n + 2$

De $c[2] = 0$, on tire $k[2] = A(3, k[3] - 1)$ et

$$v = A(2, A(3, k[3] - 1)) = A(3, k[3]).$$

Par récurrence, on montre que

$$v = A(u, k[u])$$

et l'on retrouve D avec

$$D : \; c[m] = n - k[m]$$
$$c[j] = A(j+1, k[j+1] - 1) - k[j] \quad u + 1 \leqslant j \leqslant m - 1$$
$$c[u] = A(u+1, k[u+1] - 1) - k[u] - 1$$
$$c[u-1] = A(u, k[u])$$
$$c[1:u-2] = 1$$

qui est bien de la forme générale avec un nouveau vecteur k'

$$k'[u+1:m] = k[u:m] \quad k'[u] = k[u] + 1 \quad k'[1:u-1] = 0$$

Montrons comment construire la *preuve de l'arrêt du programme.* Nous définissons la quantité

$$f = A(m,n) - \sum_{i=1}^{m} c[i]$$

et supposons que $f = a$ en A. Si $u \neq 0$, alors $c[u] = 0$. On fait $c[u] \leftarrow 1$, de sorte que la somme des $c[i]$ croît de 1, et f devient $f = a - 1$.

Si $u = 0$, on fait d'abord $v \leftarrow c[1] + 2$; $c[1] \leftarrow 0$ de sorte que la somme des $c[i]$ décroît de $c[1] = v - 2$, et donc f croît de $v - 2$. En B, $f = a + v - 2$.

Après C on fait successivement

$c[u] \leftarrow c[u] - 1$	donnant	$f = a + v - 1$
$c[u] \leftarrow v$	donnant	$f = a - 1$

Ainsi dans tous les cas on revient en A avec la valeur $a - 1$, et f décroît exactement de 1 unité à chaque pas de la boucle. On sort du programme quand

$$c[2:m] = 0 \quad , \quad c[1] = A(m,n) - 2 \quad \text{en } A, \text{ donnant } f = 2$$

Après l'initialisation $f = A(m,n) - n$. Ainsi la boucle est exécutée $A(m,n) - n - 2$ fois.

9. COMPARAISON AVEC LA PROCÉDURE DE RICE

Nous la reprenons sous la forme donnée par G. Berry [8], utilisant les deux tableaux $p[0:m]$ et $w[0:m]$, et en éliminant le cas $m = 0$

SI $n = 0$ ALORS $m \leftarrow m - 1$; $n \leftarrow 1$ SINON IS ;
$w[1:m] \leftarrow 1$; $p[1:m] \leftarrow -1$; $p[0] \leftarrow 0$; $v \leftarrow 1$;
{ « A' » $v \leftarrow v + 1$; $i \leftarrow 0$;
{ $w[i] \leftarrow v$; $p[i] \leftarrow p[i] + 1$; $i \leftarrow i + 1$; SI $i > m$ ALORS !
SINON SI $w[i] \neq p[i-1]$ ALORS ! SINON IS IS } ;
{ SI $p[m] = n$ ALORS ! SINON IS }

A' est l'assertion : il existe u tel que

$\forall\ j : \quad 0 \leqslant j \leqslant u - 1 \qquad w[j] = A(j, p[j]) = v$
$\forall\ j : \quad u \leqslant j \leqslant m \qquad w[j] = A(j, p[j]) < v < A(j, p[j] + 1)$

Il existe une certaine similarité entre A' et C de P11. Mais le vecteur k n'est pas explicité dans P11, et la méthode de travail est radicalement différente. Il est difficile de comparer les temps de calcul, hormis pour dire qu'ils sont du même ordre de grandeur. La grande boucle de la procédure de Rice est exécutée

$A(m, n) - 1$ fois, celle de P11 $A(m, n) - n - 2$ fois, ce qui revient au même car $n + 1$ est en général négligeable devant $A(m, n)$. Les variables u et v ne se comportent pas de la même façon (v ne croît pas continuement dans P11). On ne peut que conjecturer que les temps sont voisins, faute de pouvoir expérimenter (le temps croît trop vite avec m).

G. Berry présentera une discussion de la situation de la procédure proposée ici par rapport à celle de Rice, et par rapport aux mécanismes généraux de calcul, dans un prochain article de cette revue.

10. REMARQUES FINALES

Nous avons traité ici un cas limite. On ne rencontre pas tous les jours dans la pratique, un problème de la difficulté de celui de la fonction d'Ackermann, fonction qui croît si vite que même l'expérimentation devient impossible. En ce sens, ce dossier est un exercice de style. Mais la méthode employée reste valable dans des cas moins complexes [4]. Ce qui est en cause, c'est la méthode de travail empirique du programmeur [3]. Le plus souvent, il construit un programme par la méthode « essai, détection d'erreur, correction », partant d'une ébauche, la testant sur des jeux d'essai, la dépannant. On lui demande maintenant de prouver le programme résultant, ce qui est une difficulté surajoutée, dont on doute parfois qu'il puisse la maîtriser. Pour simplifier son travail, on est prêt à sacrifier l'efficacité du programme résultant. Mais alors pourquoi vouloir écrire des programmes itératifs : le recours à des procédures récursives donne généralement bien plus facilement des programmes plus sûrs. Il en est de même pour l'emploi des langages sans affectation (Arsac [4], Ashcroft [6]).

Il est possible de travailler autrement. Partant d'une première forme de programme, récursive (comme ici), ou écrite dans un langage sans affectation (comme au paragraphe 2), ou même déjà itérative, par des transformations simples, cataloguées, on le transforme en quelque chose de plus rapide, ou de plus clair. Parfois, prenant acte de propriétés du programme, un changement plus profond est opéré. Mais bien entendu, il s'agit alors de quelque chose de beaucoup plus délicat, et qu'il faut prouver. Nous espérons avoir convaincu le lecteur que cette preuve n'est pas du tout aussi complexe que la preuve du programme complet (la preuve du théorème du paragraphe 4 est évidente, celle du programme finale beaucoup plus difficile). Il n'est pas nécessaire de prouver le programme résultant : sa validité résulte de la justesse des transformations exécutées.

On peut faire deux critiques à cette façon de travailler. Elle nécessite trop de « calculs » sur les programmes, et donc augmente les risques d'erreurs. Il est vrai que l'on peut facilement se tromper lors d'une transformation. C'est en fait une affaire d'entraînement. Les transformations mises en jeu

dans le présent dossier (encore une fois, hormis les transformations sémantiques profondes) ne sont pas plus difficiles à manipuler que les transformations remarquables que nos enfants apprennent au collège

$$a^2 - b^2 = (a + b)(a - b)$$

Elles peuvent paraître plus délicates au lecteur parce qu'il n'y est pas habitué. Par ailleurs, il est certain que l'on peut se faire aider par un ordinateur pour les réaliser (programmation assistée par ordinateur). L'expérience que j'en ai m'a montré que les risques d'erreurs de calculs au cours de transformations sont moindres que les risques d'erreurs dans une programmation directe.

La seconde critique, c'est que de toutes façons, une telle façon de travailler ne se justifie pas. La plupart des programmes, dans la vie réelle, sont suffisamment simples pour s'obtenir plus directement. En tant qu'enseignant, je n'en suis pas du tout convaincu. Trop de craintes paralysent souvent le programmeur, ou l'entraînent sur des sentiers dangereux : la peur de partir sur une idée trop peu efficace, le désir d'économiser des opérations ou des instructions, qui amène parfois à d'incroyables complications, l'impossibilité de reprendre en profondeur un programme ébauché (qui amène à corriger localement au moyen de rajouts artificiels rendant le programme illisible, impossible à maintenir ou à amender...). Savoir qu'un programme se transforme aisément libère de toutes ces contraintes inutiles.

Il est hors de question de vouloir imposer une quelconque méthode de programmation, et nous ne cherchons pas à convertir le programmeur à quelque nouvelle religion. Un outil de plus est à sa disposition : les transformations de programme. Il est efficace et puissant. Il serait dommage de ne pas l'utiliser.

RÉFÉRENCES

1. E. ASHCROFT, Z. MANNA. *The translation of GOTO programs into WHILE programs IFIP conference*, North Holland Publishing Company, 1971, p. 250-255.
2. J. ARSAC. *Langages sans étiquettes et transformations de programmes*, in *Automata, Languages and Programming, 2nd Colloquium*, J. Loeckx, Ed. Springer 1974 (Lecture Notes in Computer Science n° 14), p. 112-128.
3. J. ARSAC, *Méthodes et outils en programmation*, in *Panorama de la nouveauté informatique en France*. Colloque AFCET, 1976, p. 123-146.
4. J. ARSAC, *Nouvelles leçons de programmation*. Ouvrage à paraître. Dunod, 1977. Aussi, en version abrégée, publication de l'Institut de Programmation, Paris, 1975.
5. J. ARSAC, L. NOLIN, G. RUGGIU et J. P. VASSEUR, *Le système de programmation structurée EXEL*. Revue technique Thomson-CSF, **6**, 1974, p. 715-736.
6. E. ASHCROFT et W. W. WADGE, *Lucid, a formal system for writing and proving programs*. SIAM journal on computing, **5**, 1976, p. 336-354.

7. R. M. BURSTALL et J. DARLINGTON, *A system which automatically improves programs*, Acta Informatica, **6**, 1976, p. 41-60.
8. G. BERRY, *Bottom up computation of recursive functions*, RAIRO, **10**, 1976, n° 3, p. 47-82.
9. F. COUSINEAU, *Transformation de programmes itératifs*, in Programmation, B. Robinet, Ed., Dunod, Paris, 1977, p. 53-74.
10. R. W. FLOYD, D. E. KNUTH, *Notes on avoiding GO TO statements*. Information processing letters, **1**, 1971, p. 23-31.
11. R. W. FLOYD, *Assigning meanings to programs*. Symposia in applied mathematics, Vol. 19. Rhode Island. American mathematical society, 1967, p. 19-32.
12. S. L. GERHARDT, *Knowledge about programs. A model and case study*. IEEE conference on reliable software, Los Angeles, 1975. IEEE cat. 75 CHO 940, p. 89-93.
13. C. A. R. HOARE, *An axiomatic basis for computer programming*. Com. ACM, **12**, 1969, p. 576-580.
14. J. IRLIK, *Translating some recursive procedures into iterative schemes*, in Programmation, B. Robinet, Ed., Dunod, Paris, 1976, p. 39-52.
15. D. E. KNUTH, *Structured programming with GO TO statements*, ACM computing survey, Vol. **6**, n° 4, déc. 1974.
16. R. KOSARAJU, *Analysis of structured programs*. Journal of computer ans systems science, **9**, 1974, p. 232-255.
17. H. LEDGARD et M. MARCOTTY, *A genealogy of control structures*. Com. ACM, **18**, 1975, p. 629-639.
18. Z. MANNA, *Mathematical theory of computation*, Addison Wesley, 1974.
19. L. NOLIN et G. RUGGIU, *Formalization of Exel*. ACM symposium on principles of programming languages, Boston, oct. 1973, p. 108-119.
20. W. W. PETERSON, T. KASAMI et N. TOKURA, *A on the capabilities of WHILE, REPEAT and EXIT statements*.
21. H. G. RICE, *Recursion and iteration*, Com. ACM, **8**, 1965, p. 114-115.
22. J. VUILLEMIN, *Proof techniques for recursive programs*. PhD thesis Computer science department Stanford University, 1973.