INTERPRETATION DE PROGRAMMES COMME LE TRAITEMENT D'ARBRES :

UN ASPECT DE LA PRODUCTION DE PROGRAMMES PAR TRANSFORMATIONS SUCCESSIVES

Pierre-Claude SCHOLL

RR n° 54 *Novembre 1976*

RESUME

La récursivité est un outil d'analyse puissant offert au programmeur pour mener à bien le processus de production de programmes. Le programmeur peut décider, sur des critères liés à l'environnement informatique particulier qu'il utilise, de transformer un programme récursif en un programme itératif. Nous présentons pour cela une stratégie basée sur l'interprétation de ces programmes comme le traitement de listes ou d'arbres virtuels déduits de la forme des programmes.

MOTS CLEFS ET PHRASES

Méthodologie de la programmation, Récursivité, Listes, Arbres, Transformations de programmes.

S O M M A I R E

INTRODUCTION

Le contenu de ce rapport s'insère dans le cadre de travaux menés dans le domaine général de la "méthodologie de la programmation". Il s'agit essentiellement d'étudier les mécanismes auxquels il est fait appel lors de l'écriture d'un programme afin de pouvoir mieux concilier les deux aspects contradictoires qui régissent la "collaboration" entre l'homme et la machine : d'une part le caractère purement mécaniste et non empirique de l'"activité de la machine" (exécution de programmes) et d'autre part le caractère créatif et intuitif de l'activité de l'homme (production de programmes).

La programmation dite "structurée" ou "systématique" est un courant de pensée qui tente de répondre à ce problème, de manière à, en citant BAUER dans [1], passer "from the diletantism of a home-made 'trickology' to a scientific discipline". Une idée importante est de considérer la production de programmes comme un processus qui s'appuie sur une succession de transformations de textes ([1], [7]) chacun d'entre eux reflétant un certain niveau de compréhension du problème posé et utilisant un formalisme adapté à l'interlocuteur auquel il s'adresse. Dans cette optique, la récursivité joue un rôle très important dans la mesure où d'une part c'est un outil extrêmement puissant d'analyse et d'autre part c'est un formalisme particulièrement adapté à la justification des algorithmes. On peut alors constater que la récursivité est considérée par de nombreux programmeurs comme un outil "académique" n'ayant que peu d'utilisation dans la réalité pratique. Ceci est étayé par des arguments tels que :"le langage qu'on utilise ne fournit pas la récursivité", "il est très compliqué d'écrire un programme récursif", "un programme récursif est de toutes façons inefficace". Ces arguments disparaissent dès lors que l'on distingue dans la production d'un programme la partie algorithmique (analyse et formalisation d'un problème en des termes destinés à un interlocuteur humain : le programmeur qui manipule le programme statiquement) et la partie "codage" (adaptation d'un algorithme à un environnement informatique particulier et notamment utilisation d'un langage de programmation). En d'autres termes, la "controverse" à propos de la récursivité provient essentiellement du fait que la question est mal posée au départ :

Plutôt que de la considérer comme une "primitive" de tel ou tel langage de programmation, il faut envisager la récursivité comme reflétant l'un des moyens dont nous disposons pour réduire la complexité d'un problème lors de son analyse. Ce n'est que dans un deuxième temps que les questions ci-dessus, "dispose-t-on de la récursivité dans le langage utilisé", "le programme est-il efficace" doivent être posées. Si l'on décide alors de passer à une forme itérative du programme, la connaissance des techniques de transformations systématiques est indispensable. Une des motivations pour l'étude de ces transformations est justement le désir de promouvoir l'utilisation de la récursivité en fournissant des réponses réalistes aux questions que se posent les programmeurs quant à la possibilité de produire des programmes "opérationnels" à partir d'une analyse récursive de leurs problèmes.

De nombreux travaux ont été effectués sur le problème de la transformation de programmes récursifs. Les méthodes employées sont souvent basées sur des manipulations techniques (suppression de paramètres, introduction de compteurs, ...) et sur un travail d'assertions effectué sur le programme récursif (cf.[3],[10], [12]). D'autre part l'objet même de l'étude est souvent rattaché au problème de la synthèse de programmes ou en d'autres termes à la découverte de mécanismes de transformation automatique [6].Notre optique diffère dans la mesure où comme nous l'avons dit, nous nous intéressons à donner des moyens de transformation utilisablespar un programmeur (avec toute la souplesse que ceci nous donne, contre-balancée par la nécessité d'une présentation didactique).

Le principe qui nous guide est de chercher à redéfinir le problème décrit par le programme récursif donné,de manière à faire apparaître une classe de problèmes dont une solution itérative est connue. La transformation est alors une simple adaptation de l'algorithme itératif général au problème particulier déduit de la forme récursive. Plus précisément, l'enchaînement des appels récursifs engendrés par un programme récursif peut toujours être "visualisé" sous forme d'une arborescence où chaque noeud correspond à un appel engendré, la racine étant l'appel initial. Nous cherchons à définir formellement cet arbre et à interpréter le programme itératif comme un parcours de l'arbre. Ainsi la transformation consiste à déduire de manière systématique l'arbre à partir de la forme du programme récursif, et à interpréter le programme comme un traitement de de cet arbre.

La complexité de la transformation est ainsi liée à la complexité de ce traitement. Toutefois, la méthode nous paraît devoir aboutir dans de nombreux cas, étant donné le capital de connaissances acquis par la communauté informatique dans le domaine des traitements d'arbres (cf. [8] notamment).

En fait, pour revenir au processus même de la production du programme final, le recours à la récursivité est ainsi considéré comme un moyen de reconnaître une traitement d'arbre sous jacent au problème posé. Que le programme final soit sous forme itérative ou récursive est une question, mais le fait d'avoir reconnu cet arbre et le traitement associé est en soi un élément déterminant dans la maîtrise du programme. Il devient alors plus facile d'étudier la sémantique du programme, en référence aux études sur les arbres qui fournissent autant de questions à se poser sur le programme envisagé. Notamment, l'évaluation du programme peut se faire par rapport à des quantités définies sur les arbres (nombre de noeuds, nombre de feuilles, profondeur, etc...), la simplification du programme peut être guidée par la reconnaissance de cas particuliers d'arbres (arbres binaires complets, choix de noeuds fictifs ...).

Dans le cas particulier où un appel du programme récursif ne peut engendrer qu'un appel récursif au plus, l'arborescence des appels est dégénérée et correspond à une liste. Ce type de récursivité, que nous appelons "récursivité simple", fait l'objet du premier chapître. Le fait que les listes soient des structures de données de nature beaucoup plus simple que les arbres nous permet d'autant mieux de donner des schémas généraux de transformation. Le second chapitre est consacré au cas des programmes récursifs pour lequels un appel ne peut engendrer que deux appels récursifs au plus ("récursivité double"). Leur interprétation se fait par rapport aux arbres binaires. En fin du second chapitre, nous abordons sur un exemple, le problème le plus général où le programme récursif est interprété à l'aide d'un arbre n-aire. Nous montrons comment on se ramène alors au cas de récursivité double.

Chacun des chapitres comporte d'une part une définition fonctionnelle de la structure de données de référence et quelques classes de traitements associés, d'autre part un certain nombre de schémas d'équivalence illustrés par

des exemples. Le cas échéant nous présentons l'analyse qui conduit à une forme récursive du problème. Les exemples choisis sont tous des exemples "académiques" tirés de la littérature traitant de la programmation. Le choix d'exemples classiques se justifie d'une part parce que, étant connus, ils se prêtent à une position du problème simple, d'autre part pour permettre une éventuelle comparaison entre les méthodes de transformations proposées.

La description des programmes est faite à l'aide d'une notation algorithmique proche des langages de la famille ALGOL et dont l'étude est l'objet d'une recherche parallèle [7]. Nous précisons ici quelques points nécessaires à la compréhension des algorithmes.

Du point de vue de la description des objets, on utilise le concept de "mode d'ALGOL 68. On considère de plus des modes *"composée"* qui permettent de décrire des n-uplets ordonnés d'objets. Les objets constants comportent dans leurs déclarations une initialisation repérée par le mot clef *"equ"*. La valeur initiale éventuelle des objets variables est repérée par le mot clef *"init"*.

Du point de vue de la structure des programmes, la notion d'*"univers"* de variables est introduite : un univers est composé d'un ensemble de variables et de fonctions et d'actions (procédures). L'accès aux variables à l'extérieur de l'univers ne peut se faire que par l'intermédiaire de ces fonctions ou actions. En d'autres termes, l'utilisation de variables globales est restreinte à l'univers où ces variables sont déclarées.

En ce qui concerne les actions et fonctions, une règle générale est qu'elles ne peuvent pas modifier leurs paramètres éventuels. Les fonctions peuvent renvoyer plusieurs valeurs (un n-uplet de valeurs). Dans ce cas un appel de fonction prend toujours la forme d'une affectation à un n-uplet de variables : *(A,B,C...) ← nom-de-la-fonction(liste de paramètres)*. Nous utilisons le signe "←" pour désigner ce type d'affectation entre n-uplets afin de souligner que l'affectation des diverses variables se fait collatéralement. Une fonction ne peut en aucun cas modifier une variable globale. Elle peut tout au plus accéder en lecture seulement à des variables globales. Dans ce cas la fonction est rattachée à l'univers où ces variables sont déclarées.

L'effet d'une action est entièrement défini par les modifications qu'elle apporte à des variables globales. Ainsi une action ne peut être définie que dans le contexte d'un univers, celui des variables qu'elle modifie. Une action ne peut renvoyer de valeur.

Enfin, pour terminer cette très brève présentation de la notation utilisée, notons que nous utilisons la forme suivante d'itération :

itérer
  groupe d'instructions S1
arrêt : expression logique A
  groupe d'instructions S2
fitérer

S1
tantque ¬ *A faire*
  S2
  S1
ftantque