Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Description et analyse des différentes stratégies de récupération dans les systèmes de Calcul Formel : l'exemple de PAC

Olikier, Jean-François

Award date:
1991

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE
Année académique 1990-1991

Description et analyse des différentes stratégies de récupération dans les systèmes de Calcul Formel: l'exemple de PAC

Jean-François OLIKIER

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique.

Avant-propos

Je tiens à manifester ma reconnaissance à Monsieur le Professeur B. Le Charlier qui a accepté la direction de ce mémoire et qui, par ses conseils judicieux et ses encouragements, m'a permis de le mener à bien.

J'aimerais, également, remercier toute l'équipe du laboratoire LMC de l'Institut National Polytechnique de Grenoble pour son aide efficace et, plus particulièrement, Monsieur Jean-Louis Roch qui m'a encadré lors mon stage de fin d'études.

Je voudrais, enfin, remercier toutes les personnes qui ont collaboré à la réalisation et à l'impression de ce mémoire.

Table des matières

Introduction générale

Première partie : Le Calcul Formel ou Computer Algebra

Chapitre 1 : Calcul Formel

Introduction

1. Le problème de la représentation des données

2. Exemples

Deuxième partie : le système PAC

Chapitre 2 : Introduction générale au système PAC

1. PAC (parallel algebraic computing): pourquoi?

2. PAC: un environnement distribué pour le calcul formel

3. Les contraintes du problème

4. L'état actuel

4.1 Primitives de sélection de champs
4.2 Gestion des protections
4.3 Manipulation des blocs

Troisième partie : les méthodes de récupération

Introduction

Chapitre 3 : Définition et principe de la récupération

Chapitre 4 : L'algorithme basé sur le marquage et le balayage ("Mark and Sweep")

Introduction

1. L'algorithme de marquage

1.1 Algorithme récursif du marquage
1.1.1 Description logique
1.1.2 Description algorithmique
1.2 Algorithmes itératifs du marquage
1.2.1 Formulation générale d'un algorithme itératif
1.2.1.1 Première version
1.2.1.2 Deuxième version
1.2.1.3 Troisième version
1.2.1.4 Exécution de l'algorithme itératif
1.2.1.5 Preuve de correction de l'algorithme
1.2.2 Arbre de marquage et ordre de marquage
1.2.3 Un autre algorithme général itératif de marquage
1.2.4 Représentation en mémoire
1.2.5 Un algorithme itératif de marquage sans liste AUX

2. Le balayage

2.1 La gestion d'une liste des zones libres
2.2 Le compactage de la mémoire
2.2.1 Fusion des objets non marqués adjacents
2.2.2 Mise à jour des adresses de correspondance des objets marqués
2.2.3 Mise à jour des pointeurs contenus dans les objets marqués
2.2.4 Copie des objets

Chapitre 5 : L'algorithme basé sur le comptage des références (Reference Counting Algorithm)

1. La version de base

2. La version 'hybride'

Chapitre 6 : Insuffisances des algorithmes classiques (Marquage et balayage, Compteur de références)

1. L'algorithme basé sur le comptage des références ne récupère pas les structures cycliques

2. Le problème de la mémoire virtuelle

Chapitre 7 : Les algorithmes de recopie

Introduction

1. Algorithme récursif

2. Algorithme itératif

Chapitre 8 : Le problème des applications en temps réel

Introduction

1.L'algorithme de Baker

Chapitre 9 : Les algorithmes basés sur les générations

Conclusion

Quatrième partie : Implémentation et évaluation

Chapitre 10 : Implémentation

1. La structure des données

2. Les différents algorithmes

2.1 Algorithme à base de marquage et balayage
2.2 Algorithme à base de recopie

Chapitre 11 : Evaluation et conclusion générale

Introduction

1. Evaluation

1.1 Complexité
1.2 Résultats des tests

2. Conclusion générale

Bibliographie

Annexes

Description et analyse des différentes stratégies de récupération dans les systèmes de Calcul Formel: l'exemple de PAC.

Jean-François Olikier

Errata

<u>Page 42</u>
Le quatrième point de l'invariant:
"tout objet inaccessible à partir de P0 est marqué"
doit être remplacé par:
"tout objet inaccessible à partir de P0 est non marqué"

<u>Page 44</u>
La ligne:
"Taille est la taille utile de l'objet"
doit être remplacée par:
"Taille est la taille totale de l'objet".

INTRODUCTION GENERALE

Le Calcul Formel, discipline à la frontière des mathématiques et de l'informatique, a pour objet de concevoir, analyser, implémenter et appliquer des algorithmes de calcul algébrique.

Compte tenu de leur complexité, c'est sur les machines les plus puissantes que les systèmes de Calcul Formel présentent le plus d'intérêt. Leur utilisation s'assimile souvent à l'emploi d'une 'super-calculatrice' ce qui leur confère, à première vue, un caractère attrayant. En effet, il suffit de connaître quelques fonctions permettant de transcrire le problème à résoudre sous une forme très proche de sa forme mathématique et d'attendre la réponse.
Cet aspect est, cependant, terni par le fait qu'ils se présentent à l'utilisateur sous la forme d'une 'boîte noire' dont il est très difficile de contrôler le fonctionnement. A mesure que le problème posé se complexifie, soit par les données, soit par la fonction demandée, il devient de plus en plus important de connaître le principe de fonctionnement de ces systèmes.
Ces deux raisons - utilisation de machines très puissantes (ordinateurs parallèles) et volonté de disposer d'un système ouvert - ont abouti à la conception de PAC (Parallel Algebraic Computing). PAC est une librairie d'algorithmes parallèles pour le Calcul Formel. Il a été développé au sein du Laboratoire de Modélisation et de Calcul, dirigé par le professeur Jean Della Dora, de l'Institut National Polytechnique de Grenoble.

La complexité des systèmes de Calcul Formel provient du grand nombre d'opérations élémentaires à effectuer et de la lourdeur des structures de données utilisées. De plus, l'exécution des algorithmes génère un grand nombre d'objets, c'est pourquoi elle nécessite un espace de stockage sans cesse croissant. La taille de la mémoire étant limitée, il est crucial de pouvoir récupérer les zones de mémoire occupées par les objets devenus inutiles.

Le présent mémoire vise, précisément, à étudier les différents algorithmes de récupération existants afin de déterminer lequel d'entre eux est le plus approprié pour le système PAC.

Notre travail se divise en quatre parties. La première partie définit le concept de Calcul Formel, décrit le mode de fonctionnement d'un système de Calcul Formel et passe en revue les différents systèmes disponibles. La deuxième partie donne une description du système PAC et fixe les hypothèses de travail du récupérateur. Dans la troisième partie, les différentes familles d'algorithmes de récupération sont abordées. Pour chacune d'elles, nous énonçons les spécifications et donnons une version de l'algorithme. La quatrième partie, quant à elle, est consacrée à l'implémentation et à l'évaluation de deux algorithmes pouvant être implémentés dans le système PAC.

PREMIERE PARTIE

LE CALCUL FORMEL ou COMPUTER ALGEBRA

CHAPITRE 1

CALCUL FORMEL

INTRODUCTION

Calcul scientifique et calcul algébrique

Les ordinateurs ont été inventés dans les années 40' pour décoder les messages ennemis lors de la guerre 40-45. Les premières applications étaient essentiellement des applications scientifiques de type numérique. Très vite, les applications de gestion sont devenues de plus en plus importantes en terme de temps calcul alloué. Néanmoins, les applications scientifiques sont restées les plus exigeantes au niveau de la puissance de calcul requise.

La notion de 'calcul scientifique' est, d'un point de vue 'informatique', assez ambiguë. De manière générale, un calcul scientifique est en effet composé d'un mélange de calcul algébrique (calcul sur les formules mathématiques au moyen de transformations, substitutions, simplifications,...) et de calcul numérique. L'arrivée des ordinateurs a permis de réaliser de plus en plus vite des calculs numériques de plus en plus volumineux. Ceci a conduit à réduire le calcul scientifique au calcul numérique. Cependant, le calcul numérique n'élimine pas le calcul algébrique: l'écriture du moindre programme numérique nécessite toujours la mise au point de formules sur lesquelles est basé l'algorithme. [2]

Intérêt du calcul algébrique

D'un point de vue pratique, les concepts scientifiques sont souvent exprimés par des expressions mathématiques où les valeurs numériques ont peu d'intérêt.
Considérons par exemple l'expression $3\pi^2/\pi$. Il est clair que l'on peut simplifier cette expression en divisant simultanément numérateur et dénominateur par π pour obtenir l'expression simplifiée 3π.
La valeur numérique de 3π peut se révéler intéressante, mais conserver cette expression sous sa forme symbolique non numérique est souvent suffisant et même parfois plus utile.

Un ordinateur qui ne sait réaliser que du calcul numérique doit, pour stocker cette valeur symbolique, la calculer avec un certain nombre (souvent restreint) de chiffres significatifs. Si l'ordinateur calcule avec 10 chiffres significatifs, il obtiendra la valeur 9,424777958. Non seulement ce nombre constitue une suite de chiffres assez peu évocatrice de ce qu'est $3\pi^2/\pi$, mais il ne correspond pas non plus à l'évaluation numérique exacte de 3π donnée avec 10 chiffres significatifs qui est 9,424777692.
L'écart sur les 2 dernières décimales provient des erreurs d'arrondis imputables à l'ordinateur.
Dans ces conditions, il est peu probable que l'ordinateur puisse repérer l'équivalence de $3\pi^2/\pi$ et de 3π. [3]

Trois avantages des systèmes de calcul algébrique peuvent être mis en évidence:

- *l'ordinateur qui aurait d'abord simplifié l'expression algébrique avant de l'évaluer numériquement aurait été plus rapide,*

- *à la différence des approximations faites lors de l'évaluation numérique, les réponses algébriques sont exactes (par définition, les approximations introduisent nécessairement des erreurs et, dans une suite assez longue d'opérations numériques, les erreurs s'accumulent pour donner un résultat final erroné. Le résultat final d'une suite de calculs numériques ne s'apprécie qu'au vu du résultat du calcul d'erreur associé; or le calcul d'erreurs est l'un des problèmes les plus complexes que l'on rencontre dans bien des domaines),*

- *un résultat sous forme algébrique répond souvent mieux aux desseins de la recherche scientifique ("un ordinateur doit aider à comprendre et non à calculer"; Richard Hamming; Bell Laboratories).* [3]

Le Calcul Formel

Le Calcul Formel (ou Computer Algebra) est une discipline scientifique qui *conçoit, analyse, implémente et applique des algorithmes de calcul algébrique.*

Ces algorithmes se caractérisent par des spécifications formelles simples, l'existence de preuves de correction et des bornes asymptotiques de temps d'exécution qui peuvent être établies sur base d'une théorie mathématique bien développée.

Grâce à la représentation exacte des objets algébriques dans la mémoire de l'ordinateur, les calculs algébriques peuvent être exécutés sans perte de précision ou de signification. [1]

Un environnement d'aide au calcul algébrique doit être un système complet comprenant:

- *une méthode de représentation de données non numériques d'une structure très spéciale,*

- *un langage permettant de les manipuler,*

- *une bibliothèque de fonctions efficaces pour effectuer les opérations algébriques de base qui sont nécessaires.* [2]

Les systèmes de Calcul Formel

Il existe de nombreux systèmes de Calcul Formel, mais les plus puissants et les plus connus ne dépassent guère la dizaine.
Citons pour exemples: MACSYMA, REDUCE, DERIVE, MAPPLE, MATHMATICA, SCRATCHPAD II.

- MACSYMA est le système de Calcul Formel le plus complet mais n'est disponible que sur un nombre limité de types d'ordinateurs,

- REDUCE est le système de Calcul Formel le plus diffusé, sa syntaxe familière de type PASCAL rendant son apprentissage et son utilisation faciles. *Les principales possibilités offertes à l'utilisateur sont:*
  - *l'arithmétique entière (et rationnelle) à précision 'infinie' c'est-à-dire avec un nombre arbitraire de chiffres,*
  - *l'algèbre des polynômes à une ou plusieurs variables (PGCD, factorisation de polynômes à coefficients entiers),*
  - *l'algèbre des matrices à coefficients polynômiaux ou symboliques (déterminants, résolution de systèmes linéaires, inversion),*
  - *l'analyse (différentiation, intégration),*
  - *la manipulation des expressions (simplification, substitution)* [2]

- DERIVE est un des systèmes les plus facilement disponibles sur micro-ordinateur, ce qui lui assure une grande diffusion, mais son utilisation est plutôt limitée et le réserve surtout à l'initiation,

- MAPPLE est un système relativement récent et assez complet. Il est écrit en C, ce qui le différencie des autres qui sont souvent écrits en LISP. Il utilise abondamment les techniques du "windowing" ce qui le rend plutôt agréable à utiliser. Il tourne sur Macintosh ou sur station X.

- MATHEMATICA

- SCRATCHPAD II est le système le plus récent. Il a été conçu par IBM et est encore au stade d'expérimentation (il n'est pas commercialisé, IBM le prête aux différents laboratoires de recherche en Calcul Formel afin qu'ils le testent et le mettent à l'épreuve. Le laboratoire LMC de l'Institut National Polytechnique de Grenoble, lieu de mon stage, dispose d'une version de ce système qui tourne sur une station IBM PC RT).

Certains de ces systèmes permettent une utilisation interactive par le biais de commandes, d'autres s'exécutent en processus "batch", enfin, certains combinent les deux. [1]

En général, les systèmes de Calcul Formel ne sont utilisables que sur grosses machines ou stations de travail dont le système d'exploitation fonctionne en mémoire virtuelle: pour travailler confortablement, une mémoire de travail d'un méga-octet semble être un minimum raisonnable.

DERIVE et muMATH peuvent tourner sur compatibles PC;

MAPPLE et MATHEMATICA peuvent être exécutés sur Apple Macintosh et stations de travail UNIX;

MACSYMA fonctionne uniquement sur VAX et certaines stations de travail (Symbolics, Sun);

REDUCE peut tourner sur tous les modèles de grosses machines (VAX, IBM, Cyber,...) ou mini-ordinateurs;

SCRATCHPAD ne fonctionne que sur quelques machines IBM.

En général, les systèmes de Calcul Formel présentent les propriétés suivantes:

- *la programmation est principalement interactive: l'utilisateur ne connaît, en principe, ni la forme, ni la taille de ses résultats et doit donc pouvoir intervenir à tout moment,*

- *la plupart des données manipulées sont des expressions mathématiques dont au moins la représentation externe est celle à laquelle tout le monde est habitué,*

- *le langage d'utilisation est souvent du type ALGOL,*

- *le langage d'implantation est souvent LISP; en tout cas, les données sont des listes chaînées et des arbres, et la gestion de la mémoire est dynamique avec récupération automatique de la place disponible.* [2]

Utilisation des systèmes de Calcul Formel

L'utilisation des systèmes de Calcul Formel pour des problèmes simples est relativement facile. En effet, il suffit de connaître quelques fonctions permettant de transcrire le problème à résoudre sous une forme très proche de sa forme mathématique et d'attendre la réponse.

A mesure que la taille des données s'accroît ou que les fonctions demandées se complexifient, il devient toutefois nécessaire de se faire une idée du principe de fonctionnement du système et notamment de la manière dont les données sont représentées et gérées. La connaissance des principes de fonctionnement permet d'estimer la durée des calculs et si nécessaire, de les optimiser. *Ces estimations sont en effet essentielles: pour beaucoup de calculs formels, les résultats sont quasi-instantanés, et tout va très bien. Mais si ce n'est pas le cas, la croissance de la durée et de la place mémoire nécessaires sont généralement exponentielles. Ainsi, la faisabilité d'un calcul donné n'est pas toujours évidente, et il est absurde d'engager des crédits qui peuvent être importants quand on peut prévoir un échec.* [2]

De plus, des démarches différentes au niveau de la modélisation du problème et de la méthodologie suivie lors de la programmation peuvent générer des écarts gigantesques au niveau du temps calcul ou de la mémoire consommée. *Une modélisation intelligente du problème et une programmation adaptée à la structure de données peut rendre facile un calcul autrement infaisable. L'acquisition d'un style de programmation efficace et la capacité à prévoir la taille d'un calcul sont donc bien plus importantes ici qu'en calcul numérique où les croissances sont généralement linéaires.* [2]

On remarquera que les environnements LISP et les systèmes de Calcul Formel présentent le même aspect conversationnel, les mêmes possibilités de définition de procédures ou de fonctions et donc les mêmes possibilités d'enrichissement des bibliothèques existantes sur un sujet donné. C'est la raison majeure pour utiliser les techniques de récupération des langages "LISP" dans les systèmes de Calcul Formel.

La syntaxe des langages associés à ces systèmes de calcul formel est plus ou moins celle des langages de la famille ALGOL. On retrouve donc des instructions d'affectation, un mécanisme d'appels de fonctions (commandes), un ensemble plus ou moins riche de structures de contrôle (if, do, while, repeat, etc ...), des possibilités de déclaration de fonctions ou procédures, etc ... [2]

Les domaines couverts par les systèmes de calcul formel sont:

- *les opérations sur les entiers, les rationnels, les réels et les complexes en précision illimitée,*

- *les opérations sur les polynômes en une ou plusieurs variables et sur les fractions rationnelles, le calcul du PGCD, la factorisation sur les entiers,*

- *le calcul matriciel sur des matrices à éléments numériques et/ou symboliques,*

- *la dérivation de fonctions, développement en série,*

- *la manipulation des formules: substitutions, évaluations numériques, simplifications,*

- *la résolution d'équations,*

- *l'intégration formelle,*

- *le calcul des limites.* [2]

1. LE PROBLEME DE LA REPRESENTATION DES DONNEES

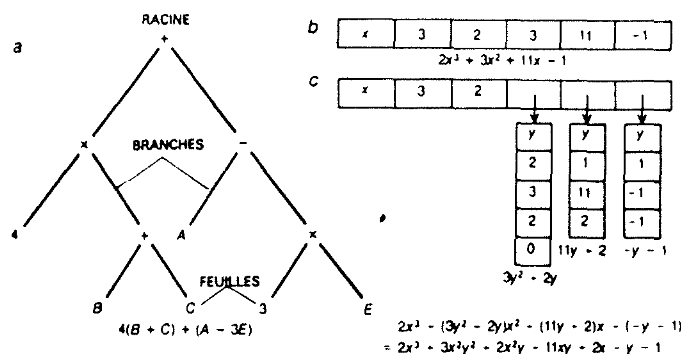
La plupart des systèmes de calcul formel représentent une expression algébrique en enregistrant l'information minimale nécessaire pour la caractériser. Deux méthodes de représentation sont habituellement utilisées.

La première consiste à représenter les expressions sous la forme d'un arbre où feuilles et noeuds représentent respectivement opérandes et opérateurs.

Dans la seconde méthode, on attribue un ordre bien défini à une suite de cases (cellules ou objets) afin d'y transposer l'information contenue dans l'expression. Cette suite de cases ou cellules constitue une liste chaînée.
Par exemple, pour représenter un polynôme à une inconnue, on place la variable dans la première cellule, le degré du polynôme (c'est-à-dire la puissance la plus élevée à laquelle est portée la variable dans le polynôme) dans la seconde cellule et les coefficients des puissances décroissantes de la variable dans les cellules suivantes.

Pour représenter des expressions plus complexes, on peut alors combiner les structures en arbre et les structures en liste chaînée.

<u>Exemple</u>



4. LA REPRÉSENTATION d'une fonction algébrique dans un ordinateur s'effectue selon différentes techniques. Le diagramme en arbre inversé (*a*) est un moyen simple et pratique de représenter la structure des liens fonctionnels entre les variables et les constantes d'une expression. On peut aussi représenter une fonction polynomiale d'une variable par un ensemble de cases (*b*) dans lesquelles on place dans l'ordre le nom de la variable, le degré du polynôme (la plus haute puissance à laquelle est élevée la variable) et les valeurs numériques des coefficients des puissances décroissantes de la variable. Pour représenter une fonction polynomiale de deux variables (*c*), on peut utiliser une généralisation de la méthode de représentation des polynômes à une variable. Dans notre exemple, un groupe de cases horizontal représente des polynômes du troisième degré en x, tandis que les groupes de cases verticaux représentent des polynômes en y. Les trois polynômes de la variable y sont ainsi les coefficients des termes du polynôme en x correspondant aux trois cases vides : le coefficient du terme en x^2 est $3y^2 + 2y$, le coefficient du terme en x est $11y + 2$ et le terme constant du polynôme en x a pour valeur $- y - 1$

source: [3]

2. EXEMPLES

Exemple général

Un énorme calcul algébrique effectué à la main en 1867 fut recalculé pour la première fois en 1970 par un système d'algèbre informatique. L'astronome français Charles Delaunay, qui mena le calcul original à bien, le publia en deux volumes. Le calcul de Delaunay, qu'il mit dix ans à faire et dix ans à vérifier, donnait la position de la Lune en fonction du temps avec une précision alors jamais atteinte. Lorsque André Deprit, Jacques Henrard et Arnold Rom des laboratoires de recherche de la société Boeing à Seatle vérifièrent le calcul par ordinateur, ils ne trouvèrent que trois erreurs dont la première était la cause des deux autres. L'ordinateur vérifia le calcul en 20 heures. [3]

Exemple d'utilisation interactive de REDUCE

Dans l'exemple, on distingue les commandes frappées par l'utilisateur des réponses du système, en mettant en caractère gras les commandes.

Quand REDUCE est lancé, il affiche un message d'invitation commençant par "REDUCE 3.3, 15-jul-87" suivit du numéro de version ainsi que le "system release".
La commande introduite, l'utilisateur termine l'expression par un point-virgule.

(x+y+z)2;**
$x^2 + 2*x*y + 2*x*z + y^2 + 2*y*z + z2$

df((x+y)3, x, 2);**
6*(x + y)

int(ea*x, x);**
$(e^a * x^2) / 2$

int(x*e(a*x)/((a*x)**2+2*a*x+1), x);**
$e^{(a*x)} / (a^2*(a*x + 1))$

for i:=1:50 product i;
30414093201713378043612608166064768844377641568960512000000000000

matrix m;
m:=mat((a,b),(b,c))$
1/m;

mat(1,1) := c/(a*c - b^2)
mat(1,2) := (- b)/(a*c - b^2)
mat(2,1) := (- b)/(a*c - b^2)
mat(2,2) := a/(a*c - b^2)

det(m);
a*c - b^2

quit;

Exemple de programme REDUCE

```
procedure LEGENDRE (N,X);
begin
	scalar S, DERIV, L, FACT;
	S := 1/(Y**2 - 2*X*Y + 1) ** (1/2);
	DERIV := df(S, Y, N);
	L := sub (Y=0, DERIV);
	FACT := for I := 1:N product I;
	return L/FACT
end;
```

DEUXIEME PARTIE

LE SYSTEME PAC

CHAPITRE 2

INTRODUCTION GENERALE AU SYSTEME PAC

1. PAC (PARALLEL ALGEBRAIC COMPUTING): POURQUOI?

La -souvent grande- complexité intrinsèque des algorithmes sophistiqués du Calcul Formel a essentiellement deux origines. D'une part, le nombre d'opérations élémentaires est important. En effet, si cette constatation n'est pas typique du Calcul Formel, on peut observer que des problèmes, même de taille restreinte, lorsqu'ils sont mal conditionnés, peuvent faire apparaître des termes intermédiaires importants qui peuvent entraîner une augmentation considérable du nombre d'opérations machine élémentaires à effectuer. D'autre part, les objets élémentaires ont une structure complexe et demandent pour leur stockage une place mémoire importante, qui ne cesse de croître en général lors de l'exécution. *La limite en stockage et en manipulation de structures de données qui souvent croissent exponentiellement, est caractéristique du Calcul Formel.* [7]

A ces deux problèmes les machines MIMD distribuées apportent une solution en autorisant l'exécution simultanée de tâches indépendantes et en augmentant, de manière générique, l'espace mémoire local nécessité pour les calculs nodaux. Par ailleurs, les systèmes de Calcul Formel classiques (Reduce, Macsyma, Maple, Sratchpad II...) offrent à l'utilisateur un système à part entière: l'utilisateur se place donc dans un système très souple et adapté à la manipulation d'objets du Calcul Formel. Mais cet aspect agréable de "super-calculatrice" présente deux inconvénients majeurs: d'une part une syntaxe et une sémantique qui lui sont propres et d'autre part une mauvaise adaptation à la communication avec l'extérieur et notamment avec les applications numériques (la génération de programmes Fortran à partir de formules algébriques est souvent le seul dialogue possible entre un programme numérique et le système de Calcul Formel).

Ces deux arguments (utilisation du parallélisme et ouverture sur des applications du calcul scientifique) ont abouti à la définition et à la réalisation de PAC.

2. PAC: UN ENVIRONNEMENT DISTRIBUE POUR LE CALCUL FORMEL

PAC (Parallel Algebraic Computing) est une bibliothèque d'algorithmes parallèles pour le Calcul Formel. Il inclut des primitives de mise au point (notamment un interpréteur) et peut être interfacé avec des systèmes de calcul formel séquentiel classiques. Une interface prototype avec Reduce a été développée dans ce sens.

Cette bibliothèque, écrite en C, présente le double avantage d'être d'une part facilement accessible depuis des codes numériques (Fortran, C, C++, ...) ce qui permet d'évaluer exactement les suites d'instructions particulièrement instables numériquement. D'autre part, il est ouvert sur l'ensemble des autres librairies disponibles (graphique par exemple).

PAC comprend actuellement:

- une bibliothèque arithmétique. Les domaines de calcul actuellement implémentés vont des entiers en précision infinie aux fractions rationnelles à plusieurs variables;

- un environnement de développement: vérification dynamique de pré- et post-conditions, interpréteur facilitant la mise au point et le test de programmes utilisant le noyau.

- un ensemble de primitives parallèles (communication des objets et traitement de certains problèmes de base).

- une interface permettant l'appel de PAC directement depuis un système de Calcul Formel classique (un prototype a été développé pour permettre l'utilisation de PAC sur FPS-T40 depuis le système séquentiel Reduce).

3. LES CONTRAINTES DU PROBLEME

La taille des objets varie dynamiquement et est souvent difficile à prédire statiquement en dehors de l'exécution effective. On ne peut donc gérer efficacement l'allocation et la libération des objets que lors de l'exécution d'un programme résolvant un problème de Calcul Formel.
Il existe donc, à ce niveau, une analogie profonde entre un tel programme et des programmes écrits dans des langages symboliques (LISP, Prolog,...): on va donc s'inspirer des techniques d'allocation et de libération utilisées dans ces langages.

Il faudra néanmoins adapter ou transformer ces algorithmes de façon à ce qu'ils s'intègrent bien dans le cadre de l'utilisation qu'en fait le système de Calcul Formel PAC. Ce cadre impose un certain nombre de contraintes:

1° il doit être conçu pour fonctionner de façon efficace sur une machine parallèle distribuée. En effet, on ne manipule sur une unité de calcul que des objets qui sont dans la mémoire propre à cette unité de calcul (principe de la localité des données). Chaque processeur connaît les objets accessibles (chez lui) à un moment donné. Les échanges entre processeurs s'effectuent par des communications.

2° on veut améliorer l'allocation (la récupération) parce que:

- la mémoire est petite: elle varie de 1 à 4 Mb pour le code et les données (or le code est de plus ou moins 200 Kb)

- on veut diminuer le temps d'allocation et de libération

- on veut faciliter la libération (implicite) tout en permettant des libérations explicites pour optimiser (éventuellement, il faut voir si cela vaut la peine d'être fait)

- les objets ont à priori une taille quelconque mais pour certains problèmes, des stratégies d'allocation spécifique peuvent s'avérer très intéressantes

Exemple

PAC a été utilisé sur un problème de factorisation, les blocs étaient de taille < 20 mots. Le temps d'allocation/libération a été nettement amélioré en n'allouant et en ne libérant que des blocs de taille 20 physiquement même si logiquement, ils étaient de tailles distinctes (< 20, exemple 2,3, etc...) (allocation/libération par une technique de free list)

4. L'ETAT ACTUEL

L'objet PAC de base est le bloc. Un bloc correspond à un objet; il est typé (le type étant inscrit dans le libellé du bloc). Lors de l'allocation, la taille du bloc (c'est-à-dire le nombre d'octets du bloc) doit être connue. L'accès aux différents champs se fait par la primitive **ChampBloc**. L'allocation se fait par **AllocBloc**. Il est essentiel de définir un type et une taille pour le bloc lors de son allocation. On peut ensuite changer cette taille: **ReallocBloc**. Un système de pile LIFO associé à un ensemble de blocs de même taille (ou de tailles du même ordre de grandeur) est utilisé pour l'allocation et la libération.
Une convention est essentielle pour les libérations mémoire: toute fonction retournant en résultat un ou des blocs doit renvoyer de "nouveaux blocs".

Exemple

Add(a,b) doit être différent de a et de b.

Cela est très limitatif: si b=0 par exemple, on est obligé de dupliquer le bloc a. Pour pallier cet inconvénient, la notion d'occurence est introduite: il est possible de faire pointer plusieurs variables sur un même bloc, sans avoir à dupliquer physiquement le bloc (**SetqBloc**).
Un bloc peut être protégé contre une libération mémoire grâce à la protection NeverFree (**ProtegerBloc**). Par exemple, le bloc représentant zéro est protégé. Lorsqu'un bloc a une occurence très grande, il passe en protection NeverFree.

Les allocations/libérations sont construites avec **malloc** et **free** du langage C mais les blocs libérés explicitement sont stockés dans un tableau de free list dont chaque entrée correspond à une plage de taille des blocs.

Il faut donner à l'utilisateur les moyens de décrire facilement la taille et la forme des objets qu'il manipule.

Les références multiples à une même donnée sont gérées par un compteur de référence associé à chaque bloc (occurrence).

copie logique -> setqbloc
copie physique -> copybloc

On dispose de deux primitives:

InitTmpSpace
FreeTmpSpace

qui permettent de faire un peu de libération implicite. Lorsqu'on rencontre un FreeTmpSpace, tous les blocs alloués après le InitTmpSpace correspondant (le dernier) sont libérés s'ils n'ont pas été référés (copiés logiquement) (occurrence nulle).

On peut mettre en évidence un certain nombre d'inconvénients. Il n'y a pas de compactage des objets actifs et seuls les objets libérés sont éventuellement fusionnés (pas encore implémentés)
=> libération "quasi" explicite.

4.1 Primitives de sélection de champs

int **LibelleBloc** (Bloc b) : renvoie le libelle du bloc b
int ***CorpsBloc** (Bloc b) : renvoie un pointeur sur le corps du bloc b
int **ChampBloc** (Blocb, int n) : renvoie le nième champ du bloc b
int **TailleBloc** (Bloc b) : renvoie la taille du bloc b
int **OccuBloc** (Bloc b) : renvoie l'occurence du bloc b
int **LibreBloc** (Bloc b) : renvoie la zone libre du bloc b
int **TypeBloc** (Bloc b) : renvoie le type du bloc b
int **ProtectionBloc** (Bloc b) : renvoie la protection du bloc b

4.2 Gestion des protections

NeverFree : bloc indestructible
SansProtection : bloc sans protection
void **ProtegerBloc** (Bloc b, int protection):
Boolean **PModifBloc** (Bloc b) : renvoie VRAI si le bloc b est modifiable

4.3 Manipulation des blocs

extern void **InitBloc** () : initialisation du gestionnaire de mémoire

void **AllocBloc** (Bloc *pt_b, int type, taille): alloue un bloc pointé par pt_b.
Le bloc est de type "type", et de taille "taille".
Si plus de place, ErreurPac est positionné à FullMemory

void **SetqBloc** (Bloc *pt_b, Bloc b): incrémente l'occurence du bloc b
L'opération correspondant à une copie logique (augmentation de l'occurence).
Si l'occurence est supérieure à MaxOccu, le bloc passe en indestructible: NeverFree

void **CopyBloc** (Bloc *pt_b, Bloc b): copie physique du bloc b
Le nouveau bloc obtenu est adressé par pt_b.

void **FreeBloc** (Bloc *pt_b): libération du bloc pointé par pt_b
si l'occurence est non nulle
alors on la décrémente
sinon si le bloc a une taille inférieure à MaxFix
alors on le met en tête de TeteFix[taille]
sinon on le libère

void **RellocBloc** (Bloc *pt_b,b; int taille): alloue un bloc pointé par pt—b et basé sur b.Le bloc est de taille "taille". Ces éléments sont identiques à ceux correspondants dans le bloc b. Si la taille de b est supérieure à taille et que b est d'occurence nulle, le bloc b est raccourci, et envoyé dans pt_b. Sinon, on renvoie dans pt_b un nouveau bloc, et si le bloc b est d'occurence nulle il est alors libéré. Si plus de place, ErreurPac est positionné à FullMemory.

void **PrintBloc** (Bloc b): impression "hardware" du bloc b
Le libellé du bloc est imprimé, puis ses différents champs.

TROISIEME PARTIE

LES METHODES DE RECUPERATION

INTRODUCTION

Après avoir présenté le système Pac, nous décrirons, dans cette troisième partie, le principe de la récupération et les différents algorithmes de récupération qui sont utilisés dans les environnements de programmation symbolique manipulant des objets de taille variable.

L'intérêt de ces algorithmes, pour les systèmes de calcul formel, réside dans le fait que les programmes de calcul formel ont le même comportement, au niveau de la création et de la destruction des objets manipulés et, d'une façon plus générale, au niveau de l'allocation dynamique de la mémoire, que les programmes écrits dans des langages de programmation symbolique.

L'ordre de présentation de ces algorithmes n'est pas le fruit du hasard. En fait, il retrace chronologiquement les différentes générations d'algorithmes.

Cette succession de générations d'algorithmes est liée à la nature des systèmes informatiques sur lesquels ils étaient implémentés. Au début de la programmation symbolique, les machines ne disposaient pas de mémoire virtuelle et les 'jobs' s'exécutaient en 'batch'. Les algorithmes de récupération basés sur le marquage et le balayage convenaient parfaitement.

Cependant, dans le but d'améliorer les performances, d'autres algorithmes furent expérimentés. C'est ainsi que des algorithmes basés sur le comptage des références furent implémentés. Malgré l'accroissement de travail lié à la gestion des tables et des compteurs de référence, ils étaient, au bout du compte, plus performants. Malheureusement, de par leur principe de fonctionnement, ils présentaient le défaut de ne pouvoir récupérer les structures cycliques. Des versions 'hybrides' ont bien été développées (algorithmes basés sur le comptage des références et, lorsque celui-ci tombe en défaut, c'est un algorithme basé sur le marquage et le balayage qui prend la relève), mais elles commençaient à ne plus être justifiées: les machines à mémoire virtuelle faisaient leur apparition.

Dans les systèmes à mémoire virtuelle, l'utilisation d'algorithmes basés sur le marquage et le balayage engendrait des temps de récupération insupportablement longs. C'est dans ce contexte qu'est apparue la première génération d'algorithmes basés sur la recopie.

Ces algorithmes présentent l'avantage de récupérer l'espace adressable sans consommer trop de temps ni trop de mémoire. Leur inconvénient est qu'ils limitent de moitié l'espace adressable par le programme.

Ces algorithmes étaient encore parfaitement adaptés jusqu'à la généralisation des applications de calcul formel (ainsi que les grosses applications LISP) en temps réel. Cette dernière évolution nous a placé dans un contexte relativement sophistiqué, mémoire virtuelle et applications en temps réel nécessitant des méthodes de récupération adaptées. *Dans la pratique, avec des espaces adressables énormes comme ceux utilisés dans les grosses applications, le temps de récupération peut atteindre plusieurs minutes voire dizaines de minutes. Il est donc utile d'avoir des algorithmes de récupération dont l'exécution peut être entrelacée avec celle du calcul normal.* [9]
Cette dernière catégorie de récupérateurs constitue la seconde génération d'algorithmes basés sur la recopie.

CHAPITRE 3

DEFINITION ET PRINCIPE DE LA RECUPERATION

Considérons le modèle simplifié d'un système informatique.
La machine peut être considérée comme étant composée de deux parties distinctes:

- un processeur contenant un certain nombre de registres ou racines. Ces derniers servent de registres hardware ou de registres de travail (de brouillon). Dans le cas d'un programme, les registres peuvent être considérés comme étant les racines d'accès à la mémoire (point d'entrée) c'est-à-dire à toutes les variables connues à un instant donné de l'exécution.
  Pratiquement, ce sont les objets référencés par des variables de l'environnement d'exécution courant.

- un espace adressable (la mémoire) qui contient un certain nombre d'objets de taille variable (en mots).

Un objet est constitué d'un certain nombre de mots mémoire contigus. Chaque mot contient soit une donnée (ou un morceau de donnée), soit un pointeur vers un autre objet. De la même façon, une racine contient soit une donnée, soit un pointeur vers un objet. Pour un objet d'un type donné, il est essentiel de savoir exactement lesquels de ses mots sont des pointeurs et lesquels sont des données. C'est possible grâce au typage des objets.

Un objet est considéré comme utile tant qu'il est accessible par le processeur ou, en d'autres termes, s'il existe un chemin issu d'une des racines (variables) qui l'atteint.

Les seuls objets qui soient accessibles par le processeur sont donc:

- cas de base: ceux qui sont référencés directement c'est-à-dire ceux qui sont adressés par les racines,

- cas inductifs: ceux qui sont référencés indirectement c'est-à-dire ceux dont l'adresse est contenue dans la partie pointeur d'un objet accessible depuis une racine.

Seuls les objets directement référencés peuvent être lus ou modifiés. Les objets indirectement accessibles peuvent devenir directement accessibles en chargeant leur adresse dans un des registres du processeur.

Une fois qu'un objet n'est plus, directement ou indirectement, accessible par le processeur, il devient inutile (ce qu'on appelle "garbage").

Lorsque qu'un objet est créé en mémoire, il est référencé par un registre du processeur. Au cours de sa vie, tout objet est référencé au moins une fois.

L'espace mémoire nécessaire à la création d'un objet est demandé par le programme de l'utilisateur, lors de son exécution, à l'allocateur de mémoire. En LISP, par exemple, ce rôle est joué par les primitives de type 'constructeur'. *Ces constructeurs ont pour fonction d'allouer une zone mémoire d'une certaine taille, d'initialiser une partie ou la totalité de cette zone et enfin de retourner une référence vers le nouvel objet alloué.* [9]

La mémoire étant constituée d'un nombre fini de mots, elle ne peut contenir qu'un nombre fini d'objets. Par conséquent, il arrive un moment où une demande d'allocation de mémoire ne peut être satisfaite par manque de mots mémoire disponibles (libres). Quand cela se produit, l'expérience montre qu'une partie des mots mémoire alloués précédemment ne sont plus utilisés (l'écrasement de pointeur est chose courante dans les langages symboliques) et peuvent donc rejoindre l'ensemble des mots mémoire libres (la liste des zones libres ou liste libre)

Une fois que l'espace adressable a été récupéré, tous les objets qui étaient accessibles le sont toujours (peut-être à un autre endroit de la mémoire, mais d'un point de vue logique c'est sans importance puisque la structure logique est inchangée). Quand aux objets inaccessibles, l'espace qu'ils occupaient a été rendu au système afin de le rendre disponible pour répondre aux demandes d'allocation ultérieures.

On peut donc définir la récupération ("Garbage Collection") comme *le processus qui consiste à récupérer la mémoire inutilisée afin de la rendre disponible à l'utilisateur (au programme) lorsque celle-ci commence à faire défaut.* [5]

Habituellement, le récupérateur est déclenché automatiquement lorsque la mémoire vient à manquer ou juste avant qu'elle ne manque. Les langages de plus haut niveau contiennent souvent des primitives qui demandent un certain nombre de mots à l'allocateur. Le récupérateur peut être déclenché lorsqu'une de ces primitives est exécutée. Par exemple, en LISP, la fonction 'cons' lance automatiquement le récupérateur. [5]

Dans certains langages comme PASCAL, c'est au programmeur de gérer l'allocation dynamique de la mémoire. Par contre dans les langages symboliques, la gestion de l'allocation de la mémoire devient vite un véritable casse-tête. Il est donc préférable de la laisser au système. C'est cette approche qui a été retenue pour le système PAC (avec en plus des possibilités de libérations explicites).

<u>Remarque</u>
Bien que la possibilité de récupération explicite soit retenue dans PAC, il est important d'avoir à l'esprit *qu'il est souhaitable que la récupération soit réalisée automatiquement pour les raisons suivantes:*

1° parce que cela dispense le programmeur de procéder lui-même à cette récupération. A ce propos, on a constaté que, quand on donne la possibilité au programmeur de récupérer explicitement des objets, on aboutissait souvent à des situations d'incohérence.

2° parce qu'une approche contrôlée par le programmeur ne saurait pas être totale dans tous les cas (c'est-à-dire récupérer tout ce qu'il est possible de récupérer). [9]

En Calcul Formel comme en programmation symbolique, l'utilisation d'un algorithme de récupération efficace est primordial:

- *l'efficacité des programmes écrits dans des langages utilisant pointeurs et records comme structures de données dépend directement de la disponibilité d'une méthode rapide de récupération* [5]

- *l'expérience fournie par de gros programmes LISP montre qu'une part importante du temps d'exécution (10 à 30 %) est dépensée en récupération* [12]

Les méthodes de récupération se décomposent généralement en deux phases:

1° identification des objets qui ne sont plus utilisés (c'est-à-dire qui ne sont plus accessibles); cette phase peut être réalisée de deux façons différentes:

- en gardant un compteur indiquant le nombre de fois qu'un objet est référencé. L'identification consiste dans ce cas à repérer les objets qui sont inaccessibles c'est-à-dire ceux dont le compteur est à zéro. On parle alors d'algorithmes basés sur le comptage des références.

- en gardant une liste des objets directement accessibles et en parcourant à partir de ces derniers tous ceux qui en sont accessibles en vue de les marquer. Habituellement, cette méthode est appelée 'marquage'.

2° rajout de cet ensemble de mots mémoire constituants les objets inutiles à la liste des zones libres; cette phase peut également être réalisée de deux façons différentes:

- en rajoutant les mots mémoire récupérés dans une liste chaînée (par pointeurs) de zones libres,

- en compactant les objets utilisés dans une même zone, l'autre partie de la mémoire étant disponible pour répondre aux demandes d'allocation.

D'autres méthodes sont également utilisées dans le cadre d'exécution de programmes sur machine à mémoire virtuelle.

Enfin, l'exécution de grosses applications en temps réel sur machines à mémoire virtuelle nécessite une adaptation des algorithmes de récupération.

Il est commode de classifier les récupérateurs selon que la taille des objets est fixe ou variable. Les premières méthodes de récupération étaient applicables uniquement pour des programmes dans lesquels tous les objets avaient la même taille.

Avec l'introduction, dans les langages de programmation, des structures de données de type 'record' (ou équivalent), il est important que les récupérateurs soient utilisables avec des objets de taille variable. Dans le cadre de ce travail, seuls les algorithmes de cette dernière catégorie seront détaillés. [5]

CHAPITRE 4

L'ALGORITHME BASE SUR LE MARQUAGE ET LE BALAYAGE

INTRODUCTION

La première grande famille d'algorithmes de récupération est basée sur le marquage de tous les objets accessibles et le balayage de la mémoire pour récupérer les objets inaccessibles ("Mark and Sweep"). [9]

Comme son nom l'indique, cet algorithme se déroule en 2 phases:
- la phase de marquage
- la phase de balayage

Dans la phase de marquage, tous les objets accessibles par le processeur sont marqués (par exemple, en mettant un bit à 1 dans l'objet). Pour ce faire, on parcourt le graphe constitué de tous les objets accessibles en partant des racines qui ne sont autres que les registres du processeur ou, à un plus haut niveau, les variables statiques de l'environnement courant d'exécution.

Dans la phase du balayage, tout l'espace adressable est parcouru. Tous les objets marqués sont visités et conservés, et leur bit de marquage est remis à 0 pour l'appel suivant au récupérateur. Les mots occupés par les objets non marqués sont rendus au système pour une allocation future. La remise des mots libres peut se faire de deux façons différentes.

La première consiste à ajouter ces mots à la liste des mots mémoire disponibles (en chaînant tous les objets inaccessibles). On obtient ainsi une liste de blocs libres qui pourront être alloués. Lors d'une demande d'allocation de mémoire, le choix d'un bloc dans la liste se fera en fonction de la taille nécessaire à la création de l'objet. Cette méthode simple présente, néanmoins, l'inconvénient d'émietter la mémoire. Au bout d'un certain temps, une demande d'allocation de mémoire ne pourra être satisfaite alors que l'espace libre total est suffisant.

Ce phénomène d'émiettement de la mémoire est encore accentué par le fait que les objets peuvent avoir des tailles très variables et que, dans les systèmes de calcul formel, les objets peuvent avoir des tailles relativement importantes (un objet de 200, voire 500, mots n'est pas chose rare).

La seconde façon de procéder consiste à compacter la mémoire lors de chaque récupération. Cette méthode revient à diviser l'espace adressable en deux parties, l'une contenant les objets accessibles (utiles) et l'autre constituant la zone de mémoire libre. Lors d'une demande d'allocation de mémoire, l'allocateur allouera les premiers mots de cette zone libre et modifiera le pointeur vers le premier mot de cette zone.

1. L'ALGORITHME DE MARQUAGE

Cette méthode de récupération, basée sur le marquage et le balayage, suppose que l'on puisse associer à chaque objet un bit supplémentaire (bit de marquage) qui est initialisé à 0. *Suivant la représentation interne des objets, ce bit de marquage peut faire partie de l'objet lui-même ou lui être associé logiquement tout en étant rangé à un emplacement physiquement séparé (cette technique, dite des "bitblocks", est utilisée dans MacLISP ["Analyse de l'implémentation de MacLISP sur Dec 20";Lisiane Goffaux; Technical-report; Université de Liège, 1980]. En effet, un mot de 36 bits étant utilisé pour stocker deux références de 18 bits, il n'y a plus de place pour un bit de marquage. On associe à chaque page un "bit block" de 16 mots comportant chacun 32 bits de marquage (ce qui donne 16*32=512=la taille d'une page) (cette approche gaspille 4 bits par mot mais vouloir les utiliser à tout prix coûterait beaucoup plus cher en temps d'exécution). Etant donné un mot, le bit block correspondant est déterminé en consultant une table avec comme index le numéro de page. Le déplacement dans la page est décomposé en deux tronçons: un de 4 bits permet de sélectionner un mot et l'autre de 5 bits permet de selectionner un bit de marquage dans le mot.* [9]

1.1 Algorithme récursif du marquage

Deux niveaux de description sont présentés: le premier est une description logique de l'algorithme tandis que le second se rapproche davantage d'un langage de programmation de type PASCAL.

1.1.1 Description logique

a. Spécifications

Effet de la procédure:
L'algorithme va marquer tous les objets accessibles à partir de la racine dont l'adresse est le paramètre passé par valeur de la procédure.

Situation initiale:
Le bit de marquage de tous les objets contenus dans la mémoire est à 0.

Situation finale:
Le bit de marquage des objets accessibles est à 1 et celui des autres objets est resté à 0.

b. Algorithme

si l'objet courant est marqué
alors {on ne fait rien}
sinon {on marque l'objet courant
et
on marque récursivement les objets pointés par les pointeurs de l'objet}

Remarques
Il est extrêmement important de marquer un objet avant de marquer ses 'constituants': cela empêche l'algorithme de boucler en présence de structures de données circulaires. [9]

1.1.2 Description algorithmique

La description détaillée de l'algorithme nécessite la définition de quelques fonctions auxiliaires:

La fonction Taille(P)
Taille(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie comme résultat la taille totale de l'objet d'adresse **P** (le nombre de mots utiles de l'objet, ainsi que ceux d'en-tête)

La fonction Taille_Utile(P)

Taille_Utile(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie comme résultat la taille utile de l'objet d'adresse **P** (le nombre de mots de l'objet, sans compter ceux d'en-tête).

Taille_Utile(**P**) = Taille(**P**) - nombre de mots d'en-tête

La fonction Mot(P,i)

Mot(P,i) est une fonction qui reçoit un pointeur **P** vers un objet et un entier **i** strictement positif et qui renvoie le **i**ème mot utile de l'objet d'adresse **P**.

La fonction Non_Marque(P)

Non_Marque(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie un booléen qui est:

VRAI si l'objet d'adresse **P** est non marqué

FAUX si l'objet d'adresse **P** est marqué.

La fonction Pointeur(M)

Pointeur(M) est une fonction qui reçoit un mot mémoire **M** et qui renvoie un booléen qui est:

VRAI si le mot **M** est un pointeur

FAUX si le mot **M** n'est pas un pointeur.

a. Spécifications

Effet de la procédure:

La procédure va marquer tous les objets accessibles à partir de l'objet d'adresse P (la racine) en modifiant le bit de marquage des objets accessibles. Le bit de marquage des objets inaccessibles reste inchangé.

Variable utilisée:

i est une variable entière qui sert à scanner les différents mots utiles d'un objet.

Situation initiale:

Le bit de marquage de tous les objets est à 0.

Situation finale:

Le bit de marquage de tous les objets accessibles est à 1 et celui des autres objets est resté à 0.

b. Algorithme

```
Début
    Si Non_Marque(P)
    Alors Début
            Marquer_objet(P)
            Pour i<- 1 Jusque Taille_Utile(P) Faire
                Début
                    Si Pointeur(Mot(P,i))
                    Alors Marquer(Mot(P,i))
                Fin
        Fin
Fin
```

Cet algorithme présente un défaut subtil. Dans un environnement de programmation où la mémoire est limitée, il se peut que nous n'ayons pas suffisamment de place que pour enregistrer la pile nécessaire aux appels récursifs. En effet, lors de chaque appel récursif, if faut créer une zone où l'on mémorise la valeur des paramètres et l'adresse de retour de la procédure.
Une version itérative est donc nécessaire.

1.2 Algorithmes itératifs du marquage

Connaissant l'adresse d'un objet, il nous est possible, en parcourant les pointeurs contenus dans cet objet, de trouver directement la liste de ses successeurs directs.

Exemple
Soit P0 l'adresse d'un objet
Nous voulons marquer tous les objets que nous pouvons atteindre par un chemin issu de P0.
Sur la figure 4.1, ci-dessous, seuls les objets hachurés doivent être marqués.

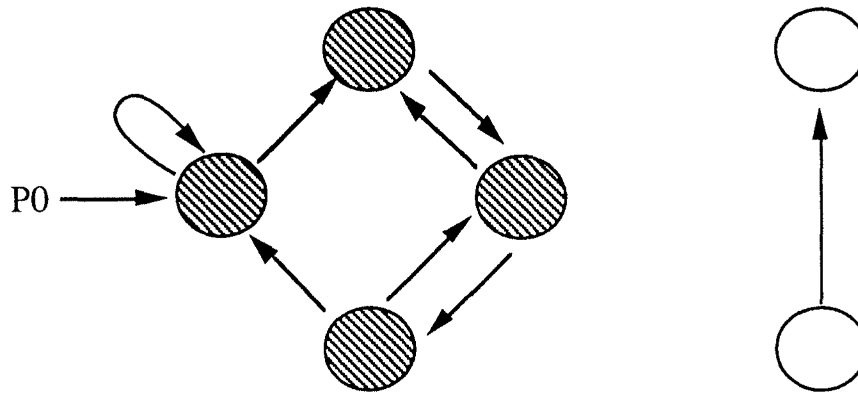


Figure 4.1

1.2.1 Formulation générale d'un algorithme itératif

Trois versions de l'algorithme itératif sont étudiées.

1.2.1.1 Première version

a. Spécifications:

Effets de la procédure:
La procédure va marquer tous les objets accessibles à partir de P0.
L'algorithme utilise une liste auxiliaire, AUX, dans laquelle on range les adresses des objets que l'on doit encore visiter.

Variables utilisées:
P est une variable de type pointeur.
AUX est la liste auxiliaire.

Situation initiale:
Aucun objet n'est marqué.
La liste AUX est vide.

Situation finale:
Tous les objets accessibles à partir de l'objet d'adresse P0 sont marqués et les autres pas. La liste AUX est vide.

b. Algorithme

Début
    AUX <- Ø
    P <- P0
L: Marquer_objet(P)
    Insérer dans la liste AUX les successeurs de objet(P)
    Tant que AUX ≠ Ø **faire**
        Début
            Extraire de la liste AUX un élément P
            Si Non_Marque(P)
            Alors aller en L
        Fin
Fin

1.2.1.2 Deuxième version

Une autre façon de formuler le même algorithme est la suivante:

a. Spécifications

L'invariant I_A est le suivant:
- AUX est la liste des adresses des objets candidats au marquage.
- L'objet d'adresse P a été marqué et les adresses de ses successeurs ont été insérées dans la liste AUX.

- tout objet accessible à partir de P0 est
  soit marqué,
  soit dans AUX,
  soit accessible à partir d'un objet de AUX via un chemin formé de sommets non marqués.
- tout objet inaccessible est non marqué.

b. Algorithme

```
Début
        AUX <- Ø
        P <- P0
        Marquer_objet(P)
        Insérer dans la liste AUX les successeurs de objet(P)
        Tant que AUX ≠ Ø Faire
A-              Début
                        Extraire de AUX un élément P
                        Si Non_Marque(P)
                        Alors Début
                                Marquer_objet(P)
                                Insérer dans AUX les successeurs de objet(P)
                              Fin
                Fin
Fin
```

1.2.1.3 Troisième version

a. Spécifications

Il est possible de reformuler l'algorithme de la façon suivante (la situation initiale est alors différente: SI = aucun objet est marqué.
L'invariant, lui, reste inchangé.

b. Algorithme

Début
AUX <- (P0)
Tant que AUX ≠ Ø **Faire**
A- **Début**
Extraire de AUX un élément P
Si Non_Marque(P)
Alors Début
Marquer_objet(P)
Insérer dans AUX les successeurs de objet(P)
Fin
Fin
Fin

1.2.1.4 Exécution de l'algorithme itératif

Illustrons l'exécution de cet algorithme sur un exemple simple schématisé à la figure 4.2.

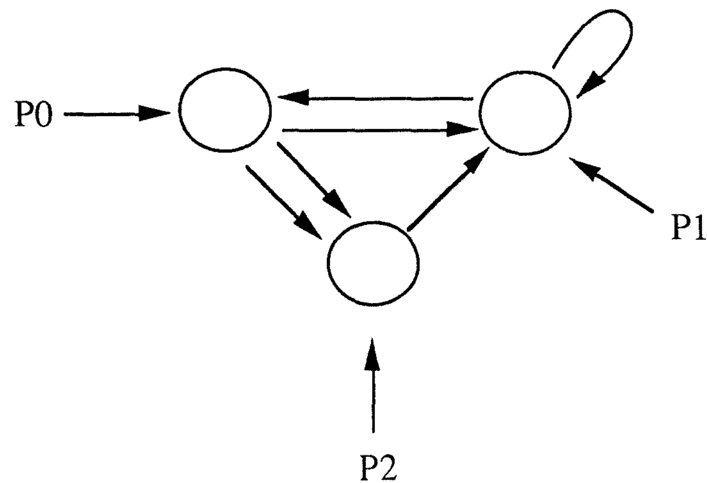


Figure 4.2

L'application de l'algorithme à la structure représentée par la figure 4.2, si l'on suppose que la stratégie de gestion de la liste est LIFO, génère les états suivants:

AUX	Objets marqués
Ø	objet(P0)
{P2, P2, P1}	
{P2, P1}	objet(P2)
{P1, P2, P1}	
{P2, P1}	objet(P1)
{P1, P0, P2, P1}	
{P0, P2, P1}	
{P2, P1}	
{P1}	
Ø	

Remarque

Cet algorithme insère dans la liste AUX les adresses des objets à visiter qu'ils soient marqués ou non. Il serait plus judicieux de n'insérer que ceux qui ne sont pas encore marqués. Cette amélioration nécessite l'ajout d'une instruction qui teste si l'adresse de l'objet qu'on est prêt à insérer dans la liste AUX correspond à une adresse d'objet déjà marqué ou non.

Plusieurs alternatives sont envisageables pour gérer la liste AUX. Nous pouvons obtenir différents ordres de marquage en changeant de stratégie pour insérer et extraire un élément. Cet aspect sera développé dans la section 1.2.2.

1.2.1.5 Preuve de correction de l'algorithme

A présent, nous allons établir la correction de l'algorithme en formulant puis en démontrant les trois propositions suivantes:

1° Tout objet marqué peut être atteint depuis P0.
2° L'algorithme se termine.
3° Après la terminaison, tous les objets accessibles depuis P0 sont marqués.

Proposition 1

Tout objet marqué peut être atteint depuis P0.

Démonstration de la proposition 1

Soit n un objet marqué.

Supposons que l'objet n soit marqué lors du $m^{ième}$ passage en L, $m \geq 1$.

On démontre par induction sur m, le nombre de passage en L.

$m = 1$: n = objet(P0) et donc n peut être atteint depuis P0.

$m > 1$: alors l'adresse de l'objet n a dû résider dans la liste AUX. Donc, n doit être un successeur d'un autre objet n' qui a été marqué avant n.
Par hypothèse d'induction, n' peut être atteint depuis P0 et donc n peut aussi être atteint depuis P0.

Proposition 2

L'algorithme se termine.

Démonstration de la proposition 2

Seuls les objets non marqués sont marqués. Comme le nombre d'objets est fini, on passera en L un nombre fini de fois. Il reste à montrer que la boucle 'tant que' ne sera pas exécutée indéfiniment.

Ceci est évident puisque un objet est extrait de la liste AUX à chaque exécution du contenu de la boucle.

Proposition 3

Après la terminaison, tous les objets accessibles depuis P0 sont marqués.

Démonstration de la proposition 3

Nous allons démontrer cette proposition par l'absurde.

Supposons que l'algorithme se soit exécuté, qu'il se soit terminé et qu'il reste un objet n, accessible depuis P0, qui ne soit pas marqué.

Soient

$$n_0 = \text{objet(P0)}, n_1, \ldots, n_p = n$$

les objets situés sur un chemin allant de objet(P0) jusqu'à n.

Soit n_s, le premier de ces objets qui ne soit pas marqué avec $s \geq 1$.
n_{s-1} est marqué et donc tous ses successeurs doivent être insérés dans la liste AUX. Au vu de l'algorithme, on voit que si, durant l'exécution, q appartient à AUX et objet(q) est non marqué, objet(q) doit devenir marqué avant la terminaison de l'algorithme sans quoi on est en contradiction avec ce qui a été dit plus haut.

1.2.2 Arbre de marquage et ordre de marquage

L'arbre de marquage est l'arbre de parcours qui se construit au fur et à mesure de l'exécution de l'algorithme. Il indique le chemin par lequel les différents objets ont été atteints durant l'exécution de la procédure de marquage.

La forme (ou la configuration) de l'arbre dépend de la stratégie adoptée pour effectuer les insertions et les extractions dans la liste AUX.

Si on se réfère à l'exemple de la figure 4.2, l'arbre de marquage obtenu lors de l'exécution de la procédure est le suivant (figure 4.3):

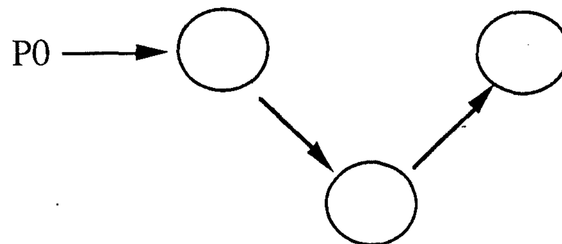


Figure 4.3

Si on avait extrait les éléments à partir de la droite, au lieu de la gauche comme c'est le cas dans l'exemple, on aurait obtenu l'arbre de marquage suivant (figure 4.4):

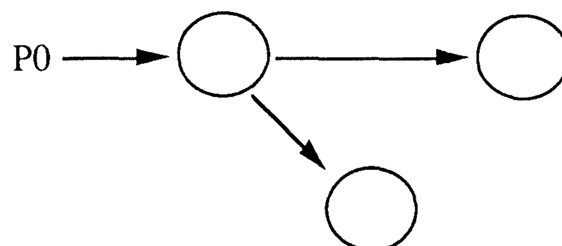


Figure 4.4

Dans la suite de ce paragraphe, nous supposerons que la structure à laquelle l'algorithme est appliqué est elle-même un arbre dont la racine est l'objet d'adresse P0.
Ceci a pour conséquence que l'arbre de marquage sera déterminé de façon unique.
Nous allons, maintenant, décrire l'ordre dans lequel les différents objets sont visités.
D'abord, on observe qu'un objet 'père' est toujours marqué avant ses 'fils' (ses successeurs).
En d'autres termes, on peut dire que la visite des objets se fait de façon préfixée (de façon à éviter les bouclages infinis en présence de structures cycliques).
La liste AUX est une suite linéaire d'éléments.
Supposons que les extractions et les insertions dans cette liste se réalisent toujours avec un seul élément à la fois.

Une **stratégie en 'pile'** pour la liste AUX signifie que les insertions et extractions se font toujours à partir du même endroit de la liste: à la fin (par exemple, à la gauche de la liste). Nous avons dans ce cas une stratégie LIFO.
Avec une stratégie en 'pile' (LIFO) pour la liste AUX, l'algorithme général itératif marque les objets dans un ordre 'en profondeur d'abord'.

Une **stratégie en 'file'** pour la liste AUX signifie que les insertions se font en une extrémité de la liste AUX (par exemple, à droite) et les extractions se font à l'autre extrémité de la liste (par exemple, à gauche). Nous avons dans ce cas une stratégie FIFO.
Avec une stratégie en 'file' (FIFO) pour la liste AUX, l'algorithme général itératif marque les objets dans un ordre 'en largeur d'abord'.

Nous allons, maintenant, décrire un autre algorithme général itératif de marquage.

1.2.3 Un autre algorithme général itératif de marquage

a. Spécifications

Un autre algorithme itératif de marquage peut être obtenu à partir d'un autre invariant.

Soit l'invariant suivant:

- AUX est la liste des objets marqués dont les successeurs n'ont pas encore été marqués. Lorsqu'un élément est extrait de la liste, l'algorithme traite (marque) directement ses successeurs (puisqu'il est traité (marqué) avant d'être inséré dans la liste AUX)
- l'objet d'adresse P, qui est extrait de la liste AUX, est marqué.
- tout objet dans AUX est marqué.
- tout objet inaccessible à partir de P0 est marqué.
- tout objet non marqué et accessible à partir de P0 est accessible à partir de AUX.

Variables utilisées

P et **Q** sont des variables de type pointeur.

b. Algorithme

```
Début
    Marquer_objet(P0)
    AUX <- {P0}
    Tant que AUX ≠ Ø faire
    Début
        Extraire un élément P de la liste AUX
        Pour tous les successeurs Q de objet(P) Faire
        Début
            Si Non_Marque(Q)
            Alors Début
                    Marquer_objet(Q)
                    Insérer Q dans AUX
                  Fin
        Fin
    Fin
Fin
```

Cet algorithme présente deux avantages par rapport au premier:

- il insère dans la liste AUX uniquement les adresses des objets successeurs, accessibles depuis P0, qui ne sont pas encore marqués.

- il permet d'être transformé aisément, en modifiant la structure de représentation des objets en mémoire, de façon à ne plus avoir besoin de mémoire pour stocker la liste AUX. Les éléments de la liste AUX seront intégrés dans la représentation des objets. Ce point sera détaillé dans la section 1.2.5.

La preuve de correction de cet algorithme étant pratiquement la même que celle du premier algorithme nous n'allons pas nous y arrêter davantage.

Intuitivement, la différence essentielle entre ces deux algorithmes est que, dans le premier, les éléments de la liste AUX sont en fait des adresses vers des objets à marquer. Ils ont une fonction de pointeur ou d'arc. Dans le deuxième algorithme, les éléments de la liste AUX doivent être considérés comme des objets dont les successeurs sont à marquer.

1.2.4 Représentation en mémoire

Supposons que la structure à marquer soit représentée en mémoire de la façon suivante.

A chaque objet est alloué une suite de mots contigus, utilisés comme sur la figure 4.5. ($n \geq 1$)

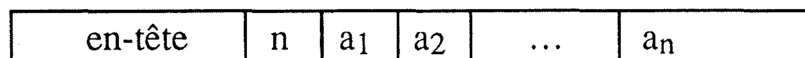


Figure 4.5

Un composant a_i peut être soit une adresse vers un autre objet, soit une donnée d'un type particulier.

La fonction **Pointeur(M)** est supposée disponible pour tester si le mot **M** est une adresse ou une donnée. Elle renvoit VRAI si **M** est une adresse et FAUX sinon.

Les algorithmes de marquage nécessitent de la place pour travailler (pour la liste AUX). Nous allons développer un algorithme dont l'espace nécessaire à la représentation de la liste fait partie intégrante de la représentation des objets. A la représentation de tout objet, il faudra ajouter un champ supplémentaire, le plus petit possible bien évidemment.

1.2.5 Un algorithme itératif de marquage sans liste AUX

a. Spécifications

Supposons que la structure d'un objet soit la suivante (figure 4.6):

Figure 4.6

où m est le bit de marquage de l'objet .

adr est le champ supplémentaire nécessaire à l'implémentation de la liste AUX.

Taille est la taille utile de l'objet .

a_1, ... , a_n sont les composants ou mots utiles de l'objet.

P0 est la racine de la structure.

Toute l'astuce de cet algorithme réside dans la façon d'implémenter la liste AUX au moyen du champ **adr** des différents objets.

Variables utilisées

x : mot mémoire

i : entier

La liste **AUX** va se répartir dans les champs **adr** des objets qui auront été atteints depuis P0.

La liste AUX sera gérée au moyen de deux pointeurs, **r** et **f**.

r contient l'adresse d'un objet dont le champ **adr** est le premier élément de la liste. Un élément de la liste est en fait un objet déjà traité (ou marqué) mais dont il faut encore traiter (ou marquer) les successeurs. Un objet est représenté par son adresse.

f contient l'adresse de l'objet dont le champ **adr** sera utilisé lors de la prochaine insertion dans la liste AUX.

Pour **insérer** un élément **x** dans la liste, il faut effectuer les opérations suivantes:

adr(f) <- x
f <- x

Pour **extraire** un élément de la liste, il faut effectuer l'opération suivante:

r <- adr(r)

La liste sera donc l'ensemble suivant:

{adr(r), adr(adr(r)), adr(adr(adr(r))), ... , f}

La stratégie de gestion de la liste est une stratégie en 'file'. Le parcours de la structure est donc un parcours en 'largeur d'abord'.

La liste AUX peut être représentée de la façon suivante (figure 4.7):

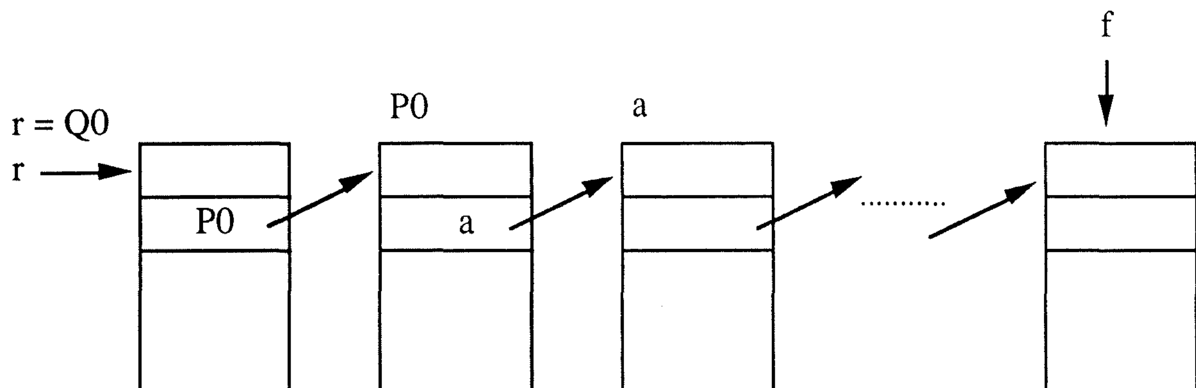


Figure 4.7

Le problème est de savoir où et comment commencer la liste AUX.
En effet, le seul objet qui soit accessible avant la récupération est la racine. C'est donc le seul objet dont le champ **adr** puisse servir pour stocker le premier élément de la liste. Or nous avons vu que le premier élément de la liste est **adr(r)**. Il est donc nécessaire de créer un objet utilitaire dont la fonction est de stocker dans son champ **adr** l'adresse de la racine, c'est à dire P0 (ceci en vue de respecter les spécifications).

r sera initialisé à l'adresse de cet objet utilitaire.
Q0 est l'adresse de l'objet utilitaire dont le champ **adr** est tel que:

adr(Q0) = P0

A la fin de l'exécution de l'algorithme, **r** contient l'adresse de l'objet dont le champ **adr** est l'adresse du dernier objet marqué, soit **R** cette dernière valeur de **r**. De ceci il découle qu'il est possible d'obtenir la liste de tous les objets accessibles par ordre de marquage. Cette liste est accessible à partir de P0:
{P0, adr(p0), adr(adr(P0)), ... , R}

En remplaçant, dans l'algorithme général itératif de marquage de la section 1.2.3, les opérations relatives:
- au marquage d'un objet,
- à l'insertion et à l'extraction d'un élément dans la liste,

nous obtenons l'algorithme suivant:

b. <u>Algorithme</u>

Début
{Marquer objet(P0)}
m(P0) <- 1

{AUX <- {P0} }
r <- Q0 {adr(r) = P0 }
f <- P0

```
{Tant que AUX ≠ Ø Faire}
Tant que r ≠ f Faire
        Début
                { extraire un élément P de la liste AUX}
                r <- adr(r)

                {pour tous les successeurs Q de objet(P) }
                Pour i <- 1 jusque Taille_Utile(r) Faire
                        Début
                                x <- Mot(r,i)
                                Si (Pointeur(x)) et (Non_Marque(x))
                                Alors Début
                                                Marquer_objet(x)
                                                adr(f) <- x
                                                f <- x
                                        Fin
                        Fin
        Fin
R <- r
Fin
```

2. LE BALAYAGE

La remise à la disposition du système des mots libérés peut s'effectuer de deux façons différentes: soit par la mise à jour d'une liste des zones libres, soit par le compactage des objets utiles dans la mémoire.

2.1 La gestion d'une liste des zones libres

Cette méthode consiste à parcourir la mémoire par ordre d'adresse croissante en partant du premier mot et à lier entre eux tous les blocs libres (objets non marqués). Les objets libres contigus sont fusionnés de façon à ne plus former qu'une seule zone dont la taille est la somme des tailles des objets constituants. La liste ainsi obtenue sera la liste des zones libres (ou liste libre). Après la récupération, toute allocation de mémoire se réalisera en cherchant, dans la liste libre, un bloc libre de taille suffisamment grande que pour pouvoir y créer l'objet voulu.

Remarque

L'allocation de mémoire devra se réaliser de façon à ne pas perturber la récupération. En effet, si une partie, seulement, d'un objet libre est allouée, il faut que l'autre partie de l'objet soit toujours présente dans la liste des zones libres et que cet objet respecte la structure des objets, à savoir la présence, dans son premier mot, de sa taille et, la présence, dans son second mot, de l'adresse du suivant ou NIL s'il est le dernier.

a. Spécifications

Effet de la procédure:

La procédure va lier entre eux les objets non marqués (libres) de façon à ce qu'ils forment une liste chaînée dont les éléments sont les zones libres de la mémoire. Les éléments de la liste sont classés par ordre d'adresse croissante des zones libres. Les objets non marqués contigus sont fusionnés de façon à ne plus former qu'un seul objet dont la taille est la somme des tailles des objets contigus.

Variables utilisées:

Free est une variable de type pointeur qui référence le premier mot de la première zone libre de la mémoire.

p est une variable de type pointeur qui sert au parcours de toute la mémoire.

Last est une variable de type pointeur qui contient l'adresse du premier mot de la dernière zone libre.

mémoire est une variable globale de type tableau dont les indices vont de 1 à Taille_Mem. Chaque élément du tableau est un mot de la mémoire. La première adresse de la mémoire est 1 et la dernière adresse est Taille_Mem. Le premier mot de chaque objet est la taille totale de l'objet. Le deuxième mot de chaque objet non marqué (libre) est soit NIL si cet objet est le dernier de la liste chaînée soit l'adresse de l'objet non marqué suivant dans la liste.

Invariants

I_A

- **Free** référence le premier mot de la première zone libre.
- **Last** référence le premier mot de la dernière zone libre.
- $1 \leq \mathbf{p} \leq$ Taille_Mem.

I_B

- **Free** référence le premier mot de la première zone libre.
- **Last** référence le premier mot de la dernière zone libre.
- $1 \leq \mathbf{p} \leq$ Taille_Mem.
- le premier mot de chaque objet est la taille totale de l'objet.
- le deuxième mot de chaque objet non marqué (libre) est soit NIL si cet objet est le dernier de la liste chaînée, soit l'adresse de l'objet non marqué suivant dans la liste.

La figure 4.8 représente la liste des objets non marqués (libres) en cours de construction.

Le premier mot de tout objet contient la taille de l'objet et le second mot de tout objet non marqué contient soit NIL si c'est le dernier objet dans la liste, soit l'adresse de l'objet suivant dans la liste.

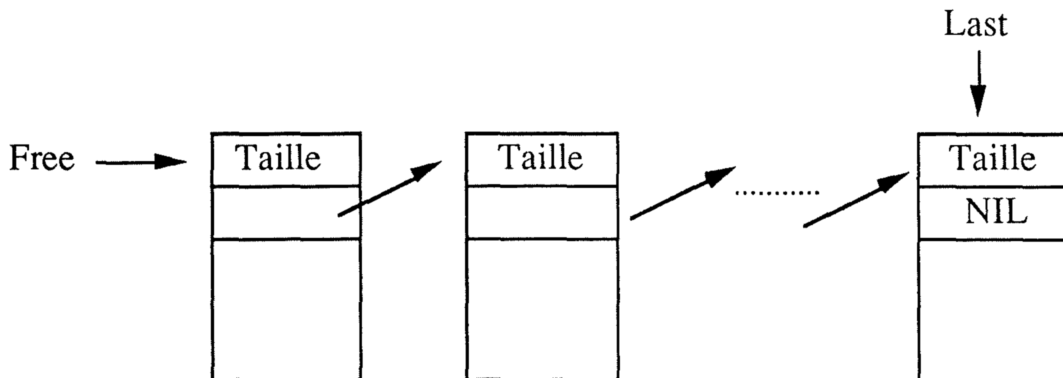


Figure 4.8

Situation initiale:
La mémoire est composée d'objets marqués (utiles) et non marqués (inutiles).

Situation finale:
Free référence la première zone libre de la mémoire et toutes les zones libres sont chaînées par ordre croissant d'adresse.

b. Algorithme

```
Début
        Free <- NIL
        Last <- NIL
        p <- 1
        Tant que ((p ≤ Taille_Mem) et (Free = NIL)) Faire
A-      Début
                Si Non_Marque(p)
                Alors Début { on est sur le premier mot d'un objet libre }
                                Free <- p
                                p <- p + Taille(p)
                        Fin
                Sinon p <- p + Taille(p)
        Fin
        Last <- Free
```

```
        {A ce moment, Free référence le premier mot de la première zone libre}
        {de la mémoire.                                                       }
        {Last référence le dernier élément de la liste des zones libres.  }

        Tant que p ≤ Taille_Mem Faire
B-      Début
              Si Non_Marque(p)
              Alors Si p = Last + Taille(Last)
                    Alors Début
                                { 2 objets non marqués se suivent}
                                Taille(Last) <- Taille(Last) + Taille(p)
                                p <- p + Taille(p)
                          Fin
                    Sinon Début
                                { les 2 objets ne se suivent pas }
                                mémoire[Last + 1] <- p
                                mémoire[p+1] <- NIL
                                Last <- p
                                p <- p + Taille(p)
                          Fin
              Sinon Début
                          { mot(i) est le premier mot d'un objet marqué }
                          p <- p + Taille(p)
                    Fin
        Fin
Fin
```

2.2 Le compactage de la mémoire

Cette méthode consiste à compacter la mémoire de façon à ce qu'elle soit composée de deux zones bien distinctes: la première zone est composée d'objets utiles tandis que la seconde est la zone de mémoire libre. Un pointeur référence le premier mot de cette zone. Le contenu du premier mot de la zone libre est la taille de cette dernière de façon à ce que la zone libre puisse être assimilée à un objet. Lors d'une allocation de mémoire, on incrémente le pointeur de zone libre et on range dans son premier mot sa taille restante. L'utilisation de cette méthode nécessite la présence d'un champ supplémentaire dans la représentation en mémoire de l'objet.

Cet algorithme se décompose en 4 phases:

1 Fusionner les objets non marqués qui sont adjacents.
2 Mettre à jour les adresses de correspondance des objets marqués.
3 Mettre à jour les pointeurs contenus dans les objets marqués.
4 Copier les objets.

2.2.1 Fusion des objets non marqués adjacents

Durant le premier passage, les zones libres adjacentes sont fusionnées de façon à ne plus former qu'une seule zone dont la taille est la somme des tailles des zones adjacentes. L'algorithme est celui qui a déjà été développé dans le paragraphe 2.1 auquel on retire les opérations liées au chaînage des objets libres. Le lien de parenté qui existe entre ces deux algorithmes est la raison pour laquelle les spécifications ne sont pas redéfinies.

Algorithme

```
Last <- NIL
p <- 1
Tant que ((p ≤ Taille_Mem) et (Last = NIL)) Faire
Début
        {soit B l'objet d'adresse i}
        Si Non_Marque(p)
        Alors Début
                        Last <- p
                        p <- p + Taille(p)
                Fin
        Sinon p <- p + Taille(p)
Fin
Tant que p ≤ Taille_Mem Faire
B-      Début
                Si Non_Marque(p)
                Alors Si p = Last + Taille(Last)
                        Alors Début
                                        { 2 objets non marqués se suivent}
                                        Taille(Last) <- Taille(Last) + Taille(p)
                                        p <- p + Taille(p)
                                Fin
                        Sinon Début
                                        { les 2 objets ne se suivent pas }
                                        Last <- p
                                        p <- p + Taille(p)
                                Fin
                Sinon Début
                                { objet(p) est un objet marqué }
                                p <- p + Taille(p)
                        Fin
        Fin
{mise à jour du premier mot de la zone libre. Le premier mot de la zone     }
{libre contient la taille de cette zone libre                               }
Memoire[Last] <- Taille_Mem - Last + 1
Fin
```

2.2.2 Mise à jour des adresses de correspondance des objets marqués

Un second passage est nécessaire pour calculer, pour chaque objet marqué, sa nouvelle adresse. Cette adresse que l'on appelle aussi adresse de correspondance ("forwarding address") est rangée dans le champ supplémentaire des objets. L'adresse de correspondance d'un objet est l'adresse à laquelle il sera copié pour compacter la mémoire (elle est l'adresse du mot mémoire qui contiendra le premier mot de l'objet (la future adresse de l'objet)).
Elle équivaut à son adresse présente moins la somme des tailles des zones libres placées avant lui dans la mémoire. Il est facile de calculer ces adresses de correspondance: on scanne la mémoire à partir du premier mot et on additionne le nombre de mots libres qu'on a déjà rencontrés. A la rencontre du premier mot d'un objet marqué, on range dans le champ correspondant à l'adresse de correspondance la différence entre l'adresse du premier mot que l'on vient de rencontrer et le nombre de mots libres que l'on a déjà parcourus.

a. Spécifications

Effet:
La procédure va calculer les adresses de correspondance de chacun des objets marqués.

Variables utilisées:
p est une variable de type pointeur.p référence le premier mot de l'objet courant.
nbre_mots_libres est une variable de type entier. Elle contient le nombre de mots libres que l'on a déjà parcourus.

Situation initiale:
Les Fwd_Add ne sont pas encore calculées

Situation finale:
Les Fwd_Add sont calculées pour tous les objets et sont stockées dans le champ Fwd_add de la représentation interne de ces derniers.

Invariant
- $1 \leq p \leq$ Taille_Mem
- les Fwd_Add des objets d'adresse < p ont déjà été calculées.

b. Algorithme

```
Début
p <- 1
nbre_mots_libres <- 0
Tant que p ≤ Taille_Mem Faire
A-    Début
      Si Non_Marqué(p)
            Alors nbre_mots_libres <- nbre_mots_libres + Taille(p)
            Sinon Fwd_Add(p) <- p - nbre_mots_libres
      p <- p + Taille(p)
      Fin
Fin
```

2.2.3 Mise à jour des pointeurs contenus dans les objets marqués

La troisième phase consiste à modifier les pointeurs. Il faut remplacer les pointeurs contenus dans les objets marqués par l'adresse de correspondance de l'objet référencé.

a. Spécifications

Effet:
La procédure va modifier les pointeurs de façon à ce qu'ils référencent toujours le même objet après le compactage.

Variables utilisées:
p est une variable de type pointeur.
i est une variable de type entier.

Situation initiale:
Les pointeurs ne sont pas modifiés

Situation finale:
Les pointeurs sont modifiés

Invariant:
- $1 \leq p \leq$ Taille_Mem
- les pointeurs contenus dans les mots d'adresse < p ont déjà été modifiés.

b. Algorithme

```
Début
p <- 1
Tant que p ≤ Taille_Mem Faire
A-   Début
     Si not (Non_Marqué(p))
     Alors Pour i <- 1 Jusque Taille_Utile(p) Faire
                Si pointeur(Mot(p,i))
                Alors memoire[p+i+1] <- Fwd_Ptr[memoire[p + i + 1]]
     p <- p + Taille(p)
     Fin
Fin
```

2.2.4 Copie des objets

La quatrième et dernière phase consiste à recopier les objets à leur adresse de correspondance ce qui aura pour effet de les copier les uns à la suite des autres. La mémoire sera ainsi compactée. C'est également au cours de cette phase que les bits de marquage des objets marqués sont remis à zéro. Après le compactage, le premier mot libre sera référencé par la variable Zone_Libre.

a. Spécifications

Effet:
La procédure va recopier les objets marqués à leur adresse de correspondance. Elle va également remettre le bit de marquage des objets recopiés à zéro pour la prochaine récupération.

Variables utilisées:
p est une variable de type pointeur
nbre_mots_libres est une variable de type entier.
Last est une variable de type pointeur qui référence le premier mot qui suit le dernier objet copié.

Free est une variable globale de type pointeur qui référence le premier mot libre de la zone libre.

Situation initiale:

Les objets marqués sont encore à leur position initiale. Le bit de marquage des objets accessibles est encore à un.

Situation finale:

Free référence le premier mot de la zone libre dont le contenu est la taille de cette dernière. En effet, comme Last n'est modifié que lors de la copie d'un objet, sa valeur après exécution de la procédure est celle qui a été calculée suite à la copie du dernier objet marqué. Elle correspond alors à l'adresse du premier mot de la zone libre, Free. Le bit de marquage des objets recopiés est remis à zéro.

Invariant:

- $1 \leq p \leq \text{Taille_Mem}$
- Last référence le premier mot qui suit le dernier objet copié

b. Algorithme

```
Début
p <- 1
nbre_mots_libres <- 0
Tant que p ≤ Taille_Mem Faire
        Début
        Si Non_marqué(p)
                Alors nbre_mots_libres <- nbre_mots_libres + Taille(p)
                Sinon Début
                                m(p) <- 0
                                Pour i <- p Jusque p - 1 + Taille(p) Faire
                                        memoire[i - nbre_mots_libres] <- memoire[i]
                                Last <- p + Taille(p) - nbre_mots_libres
                        Fin
        p <- p + Taille(p)
        Fin
Free <- Last
memoire[Free] <- Taille_Mem - Free + 1
Fin
```

CHAPITRE 5

L'ALGORITHME BASE SUR LE COMPTAGE DES REFERENCES

1. LA VERSION DE BASE

L'idée est de conserver, pour chaque objet, un compteur qui contient le nombre de références vers cet objet.

Un champ supplémentaire, appelé **réfcount**, est ajouté à la représentation de chaque objet en mémoire. Ce champ indique le nombre de fois que l'objet est référencé. Il doit être mis à jour chaque fois qu'un pointeur vers l'objet est créé ou détruit.

Lorsque le champ **refcount** devient égal à zéro, l'objet est inutile (ou inaccessible), puisqu'il n'y a plus aucun pointeur vers lui, et on peut récupérer l'espace mémoire qu'il occupe. L'espace récupéré revient au système en insérant le bloc libéré dans la liste des zones libres.

Théoriquement, **refcount** doit être suffisamment grand que pour pouvoir contenir le nombre maximum d'objets en mémoire et doit donc être suffisamment grand que pour pouvoir contenir une adresse (sachant qu'un objet à une taille minimale qui dépend de l'implémentation).

Cette méthode, bien que plus rapide que celle basée sur le marquage et le balayage, présente, néanmoins, quelques défauts. L'utilisation de cette méthode:

- nécessite l'ajout d'un champ **refcount** pour le fonctionnement du récupérateur,

- nécessite un surcroît de temps pour la mise à jour des champs **refcount**,

- ne permet pas de récupérer les structures cycliques.

L'algorithme est le suivant: chaque fois qu'un pointeur est écrit dans un registre ou dans la partie pointeur d'un objet, il modifie le champ **refcount** de l'objet anciennement référencé (refcount <- refcount - 1). Après avoir écrit le pointeur, il faut modifier (refcount <- refcount +1) le champ **refcount** de l'objet référencé par ce nouveau pointeur:

La comptabilité des pointeurs prend un temps considérable. Chaque fois qu'un pointeur est écrit, l'ancien contenu du registre ou du mot mémoire de l'objet doit être lu pour que le compteur de pointeurs de l'objet référencé par ce pointeur soit décrémenté de 1. Par contre le compteur de l'objet référencé par le nouveau pointeur doit être incrémenté de 1. Lorsque le compteur de pointeurs d'un objet atteint 0, cet objet doit être scanné pour trouver toutes les références de cet objet vers d'autres objets. Une fois que ces objets ont été déterminés, il faut décrémenter leur compteur de pointeurs de 1. Le coût en temps CPU pour cet algorithme est de plus ou moins 20%.

Une version améliorée de cet algorithme réduit le temps CPU nécessaire à la maintenance du compteur de pointeurs de tous les objets. L'algorithme diminue le temps nécessaire à la mise à jour des compteurs de pointeurs en mémorisant le compteur ailleurs que dans l'objet lui-même et en différant toutes les mises à jour pour un certain moment. Toutes les mises à jour différées sont stockées dans une table.

2. LA VERSION 'HYBRIDE'

Deutsch, Knuth et Weizenbaum ont suggéré de combiner, à la version de base, l'algorithme basé sur le marquage et le balayage. L'algorithme basé sur le comptage des références est utilisé la plupart du temps tandis que l'algorithme basé sur le marquage et le balayage, plus coûteux, est exécuté en dernier ressort. On parle alors d'algorithme hybride.

L'algorithme classique (marquage et balayage), appelé lorsque la liste des zones libres est épuisée, commence par remettre tous les compteurs de référence à zéro. Les compteurs des objets accessibles sont remis à jour, durant la phase de marquage, en incrémentant le compteur d'un objet (son champ **refcount**) chaque fois qu'il est visité.

Nous allons, maintenant, expliciter une version de la méthode 'hybride' due à Deutsch et Bobrow.

Cette méthode est basée sur la gestion de trois tables.

La première table appelée table des références multiples (Multiple Reference Table ou MRT) a comme point d'entrée une adresse d'objet et y associe le compteur de référence de l'objet. La table MRT ne contient que les adresses des objets dont le compteur de référence est supérieur ou égal à deux.

La seconde table appelée table des compteurs nuls (Zero Count Table ou ZCT) contient les adresses des objets dont le compteur de référence est à zéro.

De ceci il découle que si une adresse d'objet ne figure ni dans la table MRT, ni dans la table ZCT, alors le compteur de référence de l'objet est égal à un. En procédant de la sorte, la majorité des adresses ne figurent pas dans les tables. En effet, on peut montrer qu'une écrasante majorité des objets (97%) créés au cours de programmes LISP ne seront pas référencés plus d'une fois, et l'on peut présumer qu'il en est de même pour la majorité des programmes de calcul formel. [4]

La troisième table (Variable Reference Table ou VRT) contient les adresses des objets référencés par les variables du programme.

Deutsch et Bobrow ont observé qu'il y avait trois types d'opérations qui pouvaient modifier l'accessibilité des objets.
Ces trois types d'opérations sont:

- l'allocation d'un nouvel objet,
- l'écriture d'un pointeur,
- l'effacement d'un pointeur.

Plutôt que de mettre à jour les différentes tables lors de chaque opération, Deutsch et Bobrow ont proposé de les mémoriser dans une file. Les opérations sont traitées à intervalles de temps régulier et c'est lors du traitement de ces opérations que les trois tables sont mises à jour.

Lors d'une nouvelle allocation de mémoire pour un objet, les adresses des mots alloués sont ajoutées à la table ZCT. Cette opération est habituellement suivie de la création d'un pointeur vers cet objet. Suite à cette deuxième opération, il faut retirer de la table ZCT les adresses des mots alloués à l'objet. On constate donc qu'on défait ce qu'on vient juste de faire; autant ne pas le faire directement!

Lors de l'écriture d'un pointeur dans une variable ou un objet, des mises à jour doivent être effectuées. Trois cas sont possibles:

- Le pointeur créé référence un objet dont l'adresse est dans la table MRT. Si le compteur de référence de l'objet a atteint la valeur maximale (valeur à partir de laquelle on considère que l'objet est permanent) alors on le laisse comme tel sinon on incrémente le **refcount** d'une unité.

- Le pointeur créé référence un objet dont l'adresse est dans la table ZCT. Dans ce cas, il faut retirer l'adresse de cette table puisque son compteur de référence devient égal à un.

- Si le pointeur créé référence un objet dont l'adresse ne se trouve ni dans la table MRT ni dans la table ZCT alors c'est que la valeur du compteur de référence de cet objet était égale à un. Comme la valeur du compteur devient deux, l'adresse de cet objet doit être ajoutée à la table MRT.

Lors de l'effacement d'un pointeur (par écrasement), deux cas sont à envisager:

- Le pointeur référence un objet dont l'adresse est dans la table MRT. Le compteur de référence est décrémenté de un sauf dans le cas où il avait atteint la valeur maximale. Si la valeur du compteur, après la modification, est un alors l'adresse de cet objet est enlevée de la table MRT.

- Le pointeur référence un objet dont l'adresse n'est pas dans la table MRT. C'est que le compteur de référence de l'objet valait un et qu'il vaut à présent zéro. C'est pourquoi il faut ajouter l'adresse de cet objet à la table ZCT.

La table VRT est utilisée lors de l'ajout de nouveaux objets dans la liste des zones libres. Un objet est récupéré (ses mots sont ajoutés à la liste des zones libres) lorsque son adresse est dans la table ZCT alors qu'elle n'est pas dans la table VRT. La dernière mise à jour concerne la table ZCT: il faut enlever les adresses des objets figurant dans la table ZCT mais qui ne figurent pas dans la table VRT.

CHAPITRE 6

INSUFFISANCES DES ALGORITHMES CLASSIQUES (Marquage et balayage, Compteur de références)

1. L'ALGORITHME BASE SUR LE COMPTAGE DES REFERENCES NE RECUPERE PAS LES STRUCTURES CYCLIQUES

Dans les langages de programmation symbolique, les structures cycliques ne sont pas rares. On pourra rencontrer des objets dont le compteur de pointeurs est non nul et qui pourtant ne sont plus accessibles depuis une racine. Ces objets sont inutiles, il faut donc les rendre disponibles au système pour de futures allocations. Or, dans ce cas, l'algorithme basé sur le comptage des références tombe en défaut. Prenons l'exemple d'une liste cyclique simplement chaînée, composée de deux objets, inaccessible depuis une racine; le premier objet pointe vers le second et le second vers le premier. Dans les deux cas, le compteur de références de chaque objet est différent de zéro alors que ces deux objets sont inaccessibles depuis une racine.

Quant à l'algorithme de marquage et balayage, il détecte bien les objets inutiles mais au prix d'un travail plus conséquent.

On en déduit donc la nécessité de combiner les deux algorithmes: la majorité des demandes d'allocation devrait être satisfaite grâce à l'algorithme basé sur les compteurs de pointeurs (méthode qui est plus rapide mais pas générale) et l'utilisation de celui basé sur le marquage et le balayage ne se ferait qu'en dernier ressort.

2. LE PROBLEME DE LA MEMOIRE VIRTUELLE

Le rapport entre la taille de la mémoire secondaire et la taille de la mémoire principale est un facteur important dans la conception de récupérateur travaillant en mémoire virtuelle. Quand ce rapport est petit, l'algorithme basé sur le marquage et le balayage peut continuer à être utilisé. Par contre, lorsque ce rapport est élevé, l'introduction de nouvelles méthodes s'avère nécessaire.

L'algorithme basé sur le marquage et le balayage a été conçu initialement pour les systèmes sans mémoire virtuelle. Les récupérateurs actuels doivent aussi être efficaces sur des systèmes disposant de mémoire virtuelle.

L'utilisation de la mémoire secondaire comme mémoire virtuelle, et donc les mécanismes de pagination (page-in et page-out), change fondamentalement les hypothèses sous-jacentes à la conception d'un récupérateur. D'abord, il n'est plus nécessaire d'essayer d'éviter d'utiliser de la mémoire pour une pile puisque dans les systèmes à mémoire virtuelle, l'espace adressable disponible est considérable. Afin d'améliorer les performances des algorithmes de récupération, limiter les défauts de page, d'une part et les ralentissements dûs à un éparpillement sur plusieurs pages d'objets composant une même structure, d'autre part, deviennent des préoccupations capitales dans la conception d'algorithmes de récupération. Le compactage est donc très important dans ce type d'environnement.

Cohen et Trilling, dans [14], ont montré que la récupération associée à une étape de compactage de la mémoire apportait un gain de temps appréciable dans l'exécution de programme LISP. Ils ont également montré qu'une utilisation directe des algorithmes classiques, dans un environnement à mémoire virtuelle, aboutissait à un ralentissement insupportable de la récupération.

Le compactage, en mémoire virtuelle, ne doit pas seulement éliminer les blocs inutilisés mais doit aussi organiser la mémoire de façon telle que les pointeurs des objets référencent des objets qui leur soient voisins. [5]

Un autre point important, lié au problème de la mémoire virtuelle, concerne le working-set d'un programme. Le working-set peut être défini comme étant l'ensemble des objets activés de l'espace adressable, et donc directement accessibles par le processeur, dans le cadre de l'exécution d'un programme.

La mémoire virtuelle est divisée en pages et seulement un nombre fini de ces pages peuvent être en mémoire à moment donné. Si on fait référence à un objet qui n'est pas dans le working-set, il faut charger la page qui le contient. Ce mécanisme de chargement affecte les performances. Il est donc important de minimiser les accès aux objets qui ne sont pas dans le working-set.

Au fur et à mesure de l'exécution du programme, les objets accessibles peuvent se répartir sur tout l'espace adressable. Le problème provient du fait que le nombre de pages contenant au moins un objet est plus élévé que la taille (en pages) que peut atteindre le working-set. La solution consiste à rassembler les objets accessibles dans un nombre minimal de pages de mémoire virtuelle: cela revient à augmenter la localité des références.

L'algorithme original de marquage et balayage n'est généralement pas utilisé dans les systèmes à mémoire vituelle.

CHAPITRE 7

LES ALGORITHMES BASES SUR LA RECOPIE

INTRODUCTION

Les algorithmes de recopie sont le plus souvent utilisés dans le cas où l'espace adressable est important et où la mémoire libre est une zone contiguë [9]

La mémoire disponible (ou espace adressable) est divisée en deux parties égales appelées demi-espace. Ces deux demi-espaces sont encore appelés "oldspace" (zone ancienne) et "newspace" (zone nouvelle).

A tout instant (sauf pendant la récupération), une seule de ces deux zones, la zone ancienne, est utilisée par l'allocateur de mémoire. Au cours de l'exécution d'un programme, des demandes d'allocation de mémoire surviennent et l'allocateur de mémoire, lorsque c'est possible, les satisfait en allouant des mots mémoire de la zone ancienne ("oldspace").

Une fois qu'il n'est plus possible, pour l'allocateur de mémoire, de satisfaire une demande (soit parce que la totalité de la zone ancienne est utilisée, soit parce qu'il n'y a pas assez de mémoire pour satisfaire la demande) il est nécessaire de lancer la récupération. Durant la récupération, les objets accessibles sont transportés (ou déplacés ou copiés) dans l'autre zone (ou demi-espace) sous une forme compactée puisqu'ils sont copiés les uns à la suite des autres. Une fois que tous les objets accessibles de la zone ancienne ont été copiés dans la zone nouvelle, cette dernière prend les fonctions de l'autre et toute allocation de mémoire se réalisera dans la zone nouvelle.

Ces algorithmes consistent en fait à recopier tous les objets accessibles, à partir de l'objet 'racine' et, par conséquent, à compacter la mémoire.

Avant d'aborder plus en détail la description de ces algorithmes, illustrons tout ceci à l'aide d'un exemple.

La figure 7.1 montre la division de la mémoire en deux demi-parties: "oldspace" et "newspace". La première chose que le récupérateur doit faire, c'est copier tous les objets référencés par les racines. La racine R1 contient un pointeur vers un objet se trouvant dans "oldspace" (l'objet O1). En conséquence, l'objet O1 doit être copié dans "newspace".

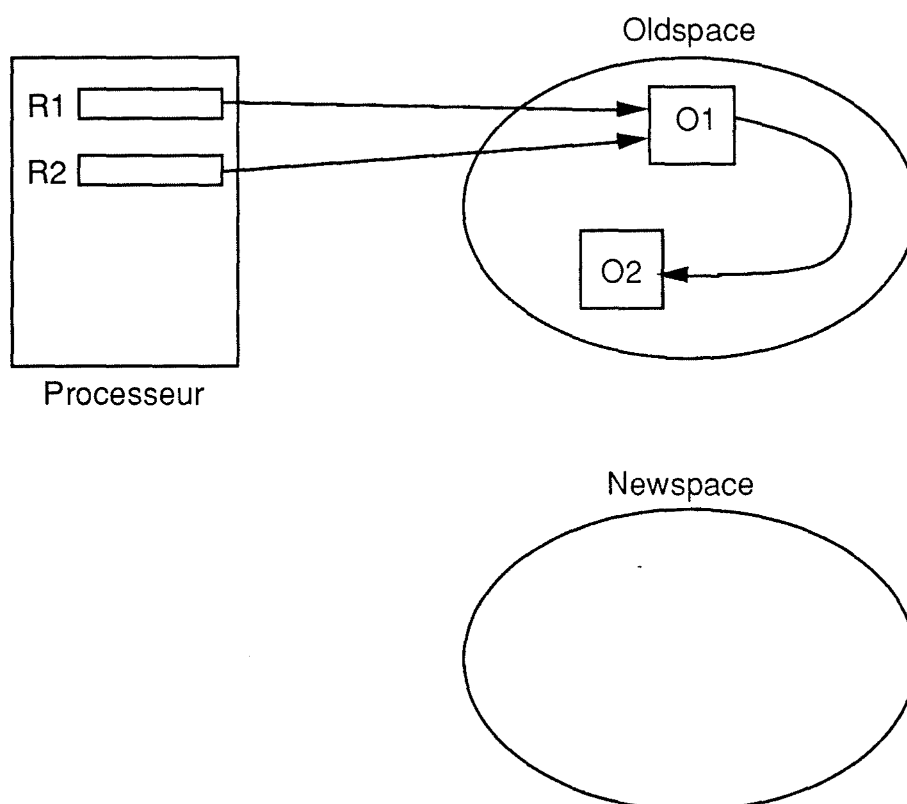


figure 7.1

A la figure 7.2, l'objet O1' est la copie exacte, dans le "newspace", de l'objet O1. Une adresse de correspondance ("forwarding pointeur") est placée dans le premier mot de l'objet original (O1). Ceci est illustré par la flèche pointillée.

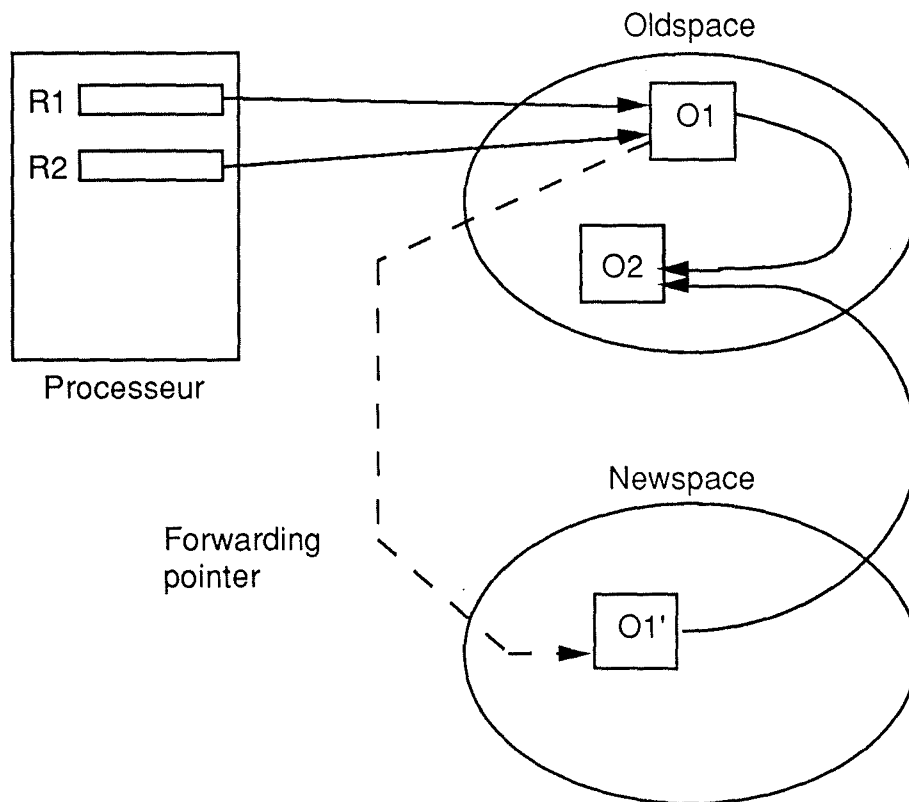


figure 7.2

Remarque

C'est ici qu'apparaît pour la première fois le besoin d'une adresse de correspondance ("forwarding pointer") propre à chaque objet. Pour la clarté de l'exposé, nous supposerons que, dans la représentation interne de tout objet, un champ supplémentaire est disponible pour contenir cette adresse de correspondance (ou "forwarding pointer"). Il est clair que le type de ce champ doit lui permettre de contenir une adresse et, par conséquent, il doit être suffisamment grand que pour pouvoir contenir la plus grande adresse que le système est en mesure de manipuler.

L'utilité de l'adresse de correspondance ("forwarding pointer") est illustrée à la figure 7.3.

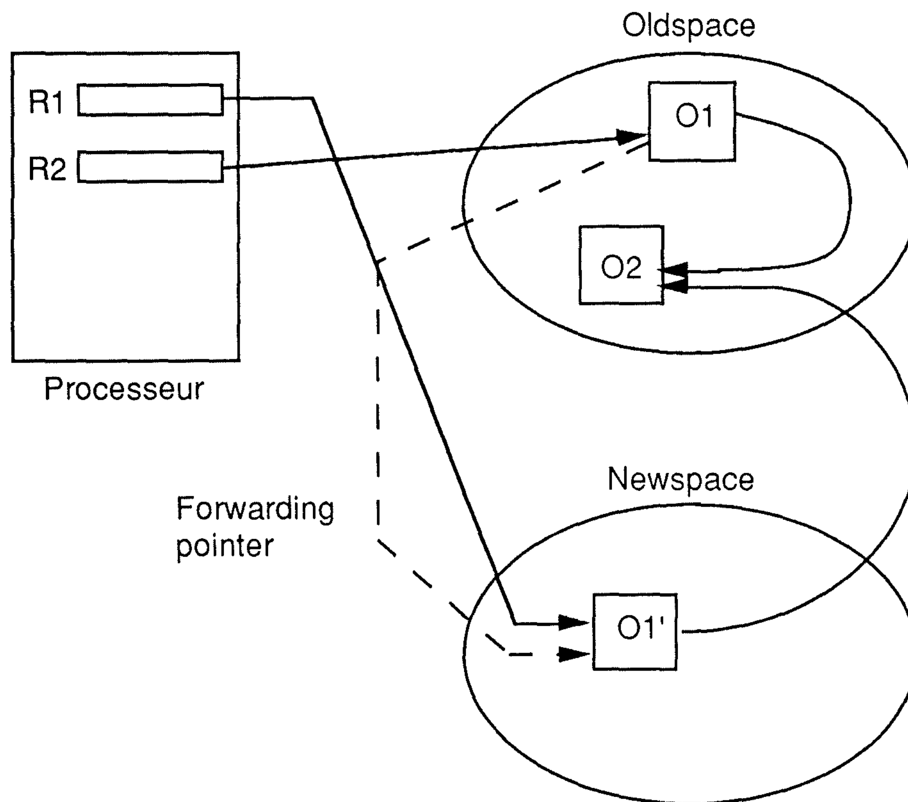


Figure 7.3

Le récupérateur veut copier l'objet référencé par la racine R2. Or il se fait que l'objet référencé par la racine R2 est encore l'objet O1. Grâce à l'adresse de correspondance, on peut savoir que cet objet O1 a déjà été copié (l'adresse de correspondance est différente de NIL) et où il a déjà été copié (cette adresse est l'adresse de correspondance). Il est inutile (ce serait une erreur grave) de copier une fois de plus l'objet O1 dans le "newspace". Il suffit de faire pointer R2 vers l'objet qui a déjà été copié dans le "newspace": c'est à dire O1'. Pour ce faire, on va remplacer la valeur de R2 par l'adresse de correspondance de l'objet O1.

Le résultat est illustré à la figure 7.4.

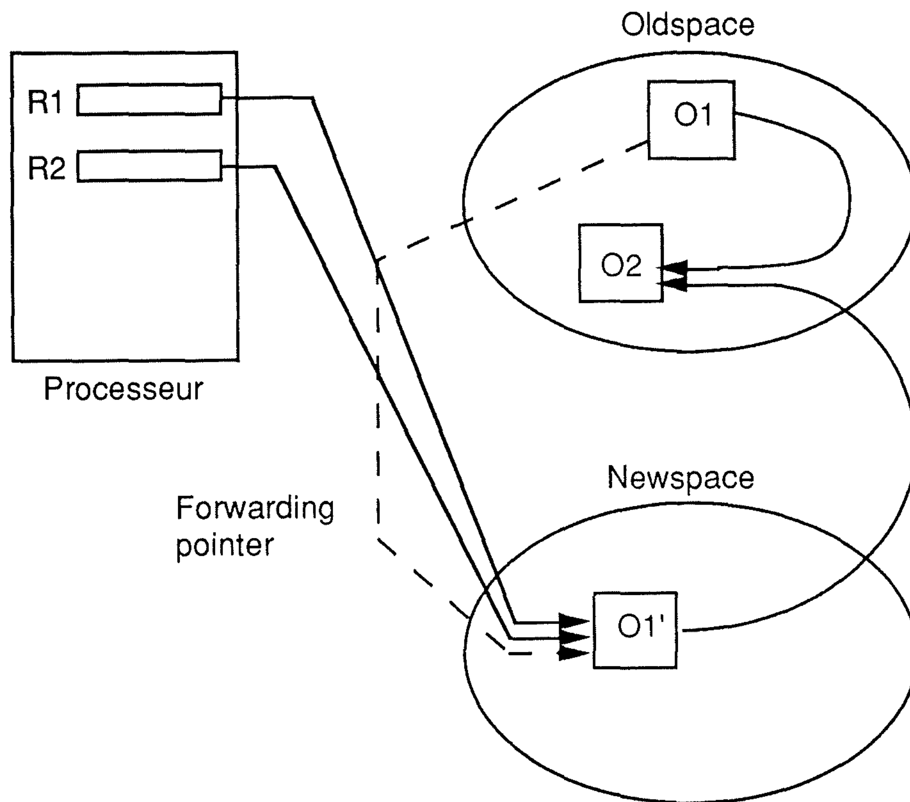


Figure 7.4

<u>Remarque</u>
Il est important d'avoir à l'esprit que *malgré les désavantages évidents que ce procédé à priori fort brutal peut présenter, le fait de tout recopier permet un certain nombre d'optimisations:*

- *vu la recopie, on assiste automatiquement à un compactage de la mémoire, ce qui, dans un environnement à mémoire virtuelle, réduit le nombre total de pages nécessaires pour représenter tous les objets accessibles à un instant donné; cette réduction du nombre total de pages est favorable à une réduction du working set donc du nombre de défauts de pages.*

- *on peut essayer, au cours de la recopie, de grouper les objets qui 'ont le plus de chance' d'être utilisés 'ensemble' comme, par exemple, les différents éléments d'une liste ou d'une structure.* [9]

  Ce type d'optimisation ne sera pas abordé dans le cadre de ce mémoire.

Revenons plus en détail aux algorithmes basés sur la recopie: dans les systèmes utilisant ce genre de récupérateur, la gestion de la mémoire est organisée comme suit:

- *L'espace adressable est divisé en deux zones de taille égale: A pour zone ancienne (ou "oldspace") et N pour zone nouvelle (ou "newspace").*

- *L'allocation dynamique se fait dans la zone A et l'exécution du programme se déroule tout à fait normalement jusqu'à épuisement de cette zone, ne s'occupant pas de la zone N.*

- *L'épuisement de la zone A provoque le déclenchement du récupérateur ("garbage collector"); il va recopier tous les objets situés dans la zone A, qualifiée de zone ancienne ou "old space", vers la zone N, qualifiée de nouvelle ou "newspace".*

- *Dès que la copie est terminée, tout l'espace situé dans la zone ancienne est récupéré et l'exécution se poursuit en ayant permuté le rôle des zones A et N. Quand la zone N est à son tour épuisée, elle sera la zone ancienne et A jouera le rôle de la zone nouvelle.* [9]

On supposera que A et N ont la même taille (ce qui permet de garantir que la place nécessaire pour la recopie est effectivement disponible)

1. ALGORITHME RECURSIF

L'algorithme peut être décrit de la façon suivante:

a. Spécifications:

- pour recopier un objet:

  - allouer, dans la zone nouvelle, la mémoire nécessaire à la représentation de l'objet;

  - recopier, dans la zone nouvelle, le contenu de la représentation, dans la zone ancienne, de l'objet;

  - écrire dans le champ "forwarding pointer" de l'ancienne représentation de l'objet la nouvelle adresse (son adresse dans la zone nouvelle)

  - recopier, récursivement, les constituants en remplaçant, au fur et à mesure, les références aux anciennes représentations par celles vers les copies.

- cas de base:

  quand on veut recopier un objet, on examine le champ "forwarding pointer" de la représentation de l'objet dans la zone ancienne:

  si le forwarding pointeur = NIL

  alors l'objet n'a pas encore été recopié et on le recopie

  sinon l'objet a déjà été recopié et sa nouvelle adresse est celle qui est rangée dans le champ "forwarding pointeur" de l'ancienne représentation.

Ceci peut encore se formuler de la façon suivante:

si le forwarding pointeur = NIL

alors begin

- allouer, dans la zone nouvelle, la mémoire nécessaire à la représentation de l'objet;

- recopier, dans la zone nouvelle, le contenu de la représentation de l'objet;

- écrire dans le champ "forwarding pointer" de l'ancienne représentation de l'objet la nouvelle adresse (son adresse dans la zone nouvelle)

- recopier récursivement les constituants en remplaçant, au fur et à mesure, les références aux anciennes représentations par celles vers les copies.

end

sinon l'objet a déjà été recopié et sa nouvelle adresse est celle qui est rangée dans le champ "forwarding pointer" de la représentation de l'objet dans la zone ancienne.
Il faut remplacer la référence à l'objet déjà copié par son adresse de correspondance.

A ce stade de la description, le formalisme est encore très discursif. Une description dans un langage plus proche d'un langage de programmation, comme PASCAL par exemple, nécessite encore quelques précisions.

Nous commencerons par supposer que la représentation de tout objet contient un champ supplémentaire, appelé "forwarding pointer", dans lequel il sera possible de stocker l'adresse de correspondance de l'objet.
Le "forwarding pointer" de tout objet respecte la propriété suivante:

soit p, un pointeur vers un objet de la zone ancienne

si forwarding_pointer(p) = NIL

alors l'objet n'a pas encore été recopié dans la zone nouvelle

sinon l'objet a déjà été recopié dans la zone nouvelle
et
forwarding_pointer(p) est son adresse dans la zone nouvelle

Pour implémenter cet algorithme de récupération, nous allons définir une fonction, la fonction **Move**, qui reçoit un pointeur p vers un objet.

Cette fonction copiera, dans la zone nouvelle, et si cela est nécessaire (en fonction de la valeur du "forwarding pointer"), l'objet référencé par le pointeur p . La fonction renverra, comme résultat, l'adresse de la représentation de l'objet dans cette zone nouvelle (l'adresse où a été recopié l'objet dans la zone nouvelle).

La récupération se poursuivra en scannant les mots constitutifs de la représentation de l'objet dans la zone **nouvelle**, de façon à identifier les mots qui contiennent un pointeur. Ces pointeurs seront remplacés par le résultat de l'application de la fonction **Move** à eux-mêmes (de façon à ce que les pointeurs contenus dans la représentation de l'objet dans la zone nouvelle correspondent à des adresses de la zone nouvelle et non à des adresses de la zone ancienne comme cela aurait été le cas si on avait scanné les mots constitutifs de la représentation de l'objet dans la zone **ancienne**).

La description de l'algorithme nécessite, pour terminer, la définition de quelques fonctions auxiliaires:

La fonction Taille(P)

Taille(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie comme résultat, la taille totale de l'objet d'adresse **P** (le nombre de mots utiles de l'objet ainsi que ceux d'en-tête (taille et "forwarding pointer"))

La fonction Taille_Utile(P)

Taille_Utile(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie comme résultat la taille utile de l'objet d'adresse **P** (le nombre de mots de l'objet, sans compter ceux d'en-tête).

Taille_Utile(**P**) = Taille(**P**) - nombre de mots d'en-tête

La fonction Fwd_Ptr(P)

Fwd_Ptr(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie comme résultat:

soit NIL si l'objet n'a pas encore été copié dans la zone nouvelle,

soit l'adresse de correspondance de l'objet, c'est-à-dire son adresse dans la zone nouvelle, s'il y a déjà été copié.

La fonction Newspace(P)

Newspace(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie un booléen qui est:

VRAI si l'adresse **P** est une adresse de la zone nouvelle

FAUX sinon

La fonction Oldspace(P)

Oldspace(P) est une fonction qui reçoit un pointeur **P** vers un objet et qui renvoie un booléen qui est:

VRAI si l'adresse **P** est une adresse de la zone ancienne

FAUX sinon

La fonction Move(P)

Move(P) est une fonction qui reçoit un pointeur **P** vers un objet. Cette fonction effectue la recopie de l'objet dans la zone nouvelle, met à jour le "forwarding pointer" de la représentation de l'objet dans la zone ancienne et renvoie comme résultat l'adresse, dans la zone nouvelle, de l'objet copié.

Variables utilisées:

Free est une variable globale de type pointeur, utilisée lors de la récupération, qui référence le premier mot libre dans la zone nouvelle.

Memoire est une variable globale de type tableau dont les indices vont de 1 à Taille_Mem. Chaque élément du tableau est un mot de la mémoire. La première adresse de la mémoire est 1 et la dernière adresse est Taille_Mem.

inter est une variable qui mémorise l'adresse où l'on a copié l'objet d'adresse P (parce que on modifie Free)

Situation initiale:
La mémoire contient des objets accessibles et des zones libres.

Situation finale:
Seuls les objets accessibles sont encore dans la mémoire. Les objets de la mémoire sont les uns à la suite des autres (forme compactée).
Free référence le premier mot de la zone libre.

b. Algorithme

```
Fonction Move(P)
Var
        inter: entier
        i: entier

Début
        Si Newspace(P)
        Alors Move <- P
        Sinon Début
                        Si Fwd_Ptr(P) = NIL
                        Alors Début
                                        inter <- Free;
                                        Free <- Free + Taille(P);
                                        Pour i <- 1 Jusque Taille(P) Faire
                                                Memoire[inter+i-1] <- Memoire[P+i-1];
                                        Memoire[P+1] <- inter
                                Fin;
                        Move <- Fwd_Ptr(P)
                Fin

        {on scanne les mots utiles de l'objet copié et lorsqu'on    }
        {tombe sur un pointeur, ptr par exemple, on le remplace          }
        {par move(ptr)                                              }
        Pour i <- 1 Jusque Taille_Utile(inter) Faire
                Si Pointeur(Memoire[inter+i+1])
                Alors Memoire[inter+i+1] <- Move(Memoire[inter+i+1])
Fin
```

2. ALGORITHME ITERATIF

La version itérative qui suit est due à Cheney [13].

a. Spécifications:

L'usage d'une pile est évité en utilisant, lors de la récupération, deux pointeurs, **ptr_free** et **ptr_scan**.

ptr_free est un pointeur qui référence le premier mot libre de la zone nouvelle. Il est initialisé à l'adresse du premier mot de la zone nouvelle. A chaque copie, il est incrémenté de la taille de l'objet copié.

ptr_scan est un pointeur qui référence l'endroit (une adresse) où en est le récupérateur dans le balayage séquentiel de la zone nouvelle. Il est également initialisé à l'adresse du premier mot de la zone nouvelle.

Ceci peut être représenté par la figure 7.5

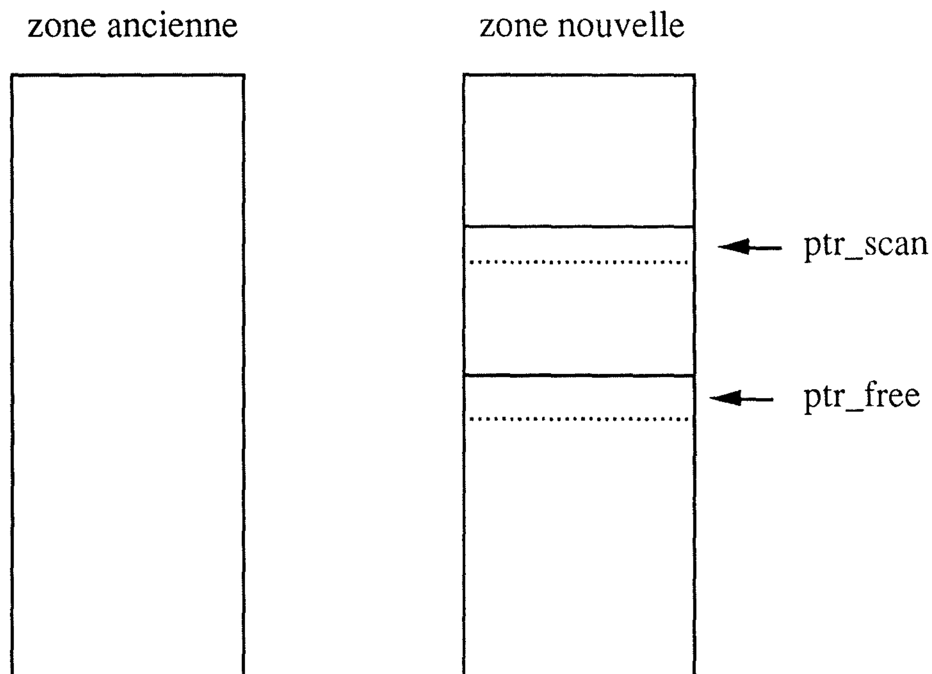


Figure 7.5

Les deux pointeurs vérifient les propriétés suivantes:

- au début de la récupération, **ptr_free** et **ptr_scan** référencent le premier mot de la zone nouvelle.

- l'espace effectivement utilisé (EEU) dans la zone nouvelle est l'ensemble des mots tels que l'adresse de ces mots vérifie la propriété suivante:
  EEU = {mot d'adresse ad tq
  première adresse de la zone nouvelle ≤ ad < **ptr_free**}

- les objets situés dans la zone [**ptr_scan** , **ptr_free** [peuvent contenir des références vers la zone ancienne

- durant la copie, début zone nouvelle ≤ **ptr_scan** ≤ **ptr_free**;
  ptr_scan ne peut être égal à **ptr_free** qu'au début et à la fin de la récupération.

La récupération se réalise en deux phases.

D'abord, ce sont les objets directement accessibles qui sont transportés (ou copiés) dans la zone nouvelle. Rappelons que par objets directement accessibles, nous entendons tous les objets dont les adresses se trouvent dans les variables déclarées statiquement. Pour la facilité de l'exposé, nous considérerons qu'il n'y en a qu'une.

Une fois que tous les objets référencés par la racine ont été copiés, le balayage de la zone nouvelle peut commencer.

La zone comprise entre les mots d'adresse **ptr_scan** et **ptr_free** contient les objets qui ont été copiés mais dont le contenu ne l'a pas encore été (objets copiés non encore explorés).Cette zone est balayée (en incrémentant **ptr_scan**).

Lors du balayage des objets déjà copiés (par **ptr_scan**), si un pointeur est trouvé, l'objet qu'il référence doit être copié dans la zone nouvelle à moins qu'il n'y soit déjà. Si l'objet référencé, par le pointeur trouvé, a une adresse de correspondance différente de NIL, alors l'objet référencé a déjà été copié et il ne faut plus le faire: il suffit alors de remplacer la valeur du pointeur trouvé (contenu dans le mot d'adresse **ptr_scan**) par l'adresse de correspondance de l'objet déjà copié.

Par contre, si l'adresse de correspondance est NIL, alors l'objet n'a pas encore été copié et il faut le copier dans la zone nouvelle et remplacer le pointeur trouvé par l'adresse de correspondance de l'objet copié. Ensuite, on incrémente d'une unité le pointeur de balayage (**ptr_scan**), de façon à visiter le mot suivant de la mémoire, et on recommence la procédure.
La procédure s'exécute jusqu'au moment où le second pointeur atteint le premier.
Lorsque les deux pointeurs sont égaux, tous les objets accessibles ont été copiés dans le "newspace" et les deux demi-espaces peuvent être permutés.

Remarque
L'ordre de recopie n'est pas le même dans le cas récursif et dans le cas itératif. En effet, dans le premier cas, les objets sont recopiés 'en profondeur d'abord' tandis que dans le second cas, ils sont recopiés 'en largeur d'abord'. Il en résultera que la localité ne sera pas la même dans les deux cas. Dans la pratique, la recopie 'en profondeur d'abord' s'avère souvent préférable. [9]

b. Algorithme

1° copier les objets référencés par les variables de l'environnement courant du programme

2° pour copier un objet pointé par p

si forward_pointer(p) = NIL

alors begin
- on copie l'objet p à partir du mot qui est pointé par **ptr_free**
- on remplace le pointeur trouvé (le mot mémoire d'adresse **ptr_scan**) par la valeur de **ptr_free**
- on donne la valeur de **ptr_free** au "forward pointer" de la représentation dans la zone ancienne de l'objet
- on incrémente la valeur de ptr_free de la taille de l'objet copié

end

sinon {forward pointer(p) contient l'adresse de l'objet dans la zone nouvelle. Il est inutile de le copier une seconde fois.}
on remplace le pointeur trouvé (contenu dans le mot d'adresse ptr_scan par la valeur de "forwarding pointer(p)"

3° On a fini de copier tous les objets lorsque les deux pointeurs sont égaux c'est-à-dire lorsqu'ils pointent tous les deux vers le même mot.

```
Début
    ptr_free <- adresse du premier mot de la zone nouvelle
    ptr_scan <- adresse du premier mot de la zone nouvelle
    Pour i <- 1 Jusque Taille(p) Faire
        memoire[ptr_free + i - 1] <- memoire[p + i - 1]
    Fwd_Ptr(p) <- ptr_free
    ptr_free <- ptr_free + Taille(p)

    Tant que ptr_free <> ptr_scan Faire
        Début
            Si pointeur(ptr_scan)
            Alors Début
                    p <- memoire[ptr_scan]
                    Si Fwd_Ptr(p) = NIL
                    Alors Début
                            Pour i <- 1 Jusque Taille(p) Faire
                                memoire[ptr_free + i - 1] <-
                                                memoire[p+i-1]
                            Fwd_Ptr(p) <- ptr_free
                            memoire[ptr_scan] <- ptr_free
                            ptr_free <- ptr_free + Taille(p)
                        Fin
                    Sinon memoire[ptr_scan] <- Fwd_Ptr(p)
                Fin
            Ptr_scan <- ptr_scan + 1
        Fin
    Free <- ptr_free
Fin
```

CHAPITRE 8

LE PROBLEME DES APPLICATIONS EN TEMPS REEL

INTRODUCTION

Jusqu'ici, seul le problème posé par l'utilisation d'environnement à mémoire virtuelle a été abordé. Nous n'avons pas encore tenu compte de l'existence d'applications en temps réel. Pourtant, ce contexte (mémoire virtuelle et applications en temps réel) mérite toute notre attention puisqu'il tend à se généraliser en Intelligence Artificielle, en Calcul Formel, etc... Il était donc nécessaire de concevoir un récupérateur permettant des temps de réponse tolérables dans un environnement interactif en temps réel.

Avant de poursuivre la description, il est utile de clarifier la notion d'application en temps réel. Une application en temps réel est une application qui présente la propriété suivante: le temps requis par chacune des opérations élémentaires est borné par une constante indépendante du nombre d'objets utilisés.[10]

Remarque
Cette constante dépend quand même de l'application et de l'ordinateur utilisé, c'est pourquoi il serait plus correct de parler de "pseudo temps réel". [10]

Nous avons supposé jusqu'ici que la récupération se déroulait de manière non interrompue, suivant le scénario suivant: lorsqu'il n'y a plus de mémoire disponible pour l'allocation d'un nouvel objet, le récupérateur entre en action et l'application est interrompue jusqu'au moment où il a fini. L'application interrompue reprend alors son exécution là où elle a été stoppée.

Le problème provient du fait que, dans la pratique, avec des espaces adressables énormes fonctionnant en mémoire virtuelle comme ceux utilisés dans les grosses applications, le temps de récupération peut atteindre plusieurs minutes voire dizaines de minutes [9]. Il est donc primordial de disposer de méthodes de récupération dont l'exécution puisse être entrelacée avec celle de l'application.

1. L'ALGORITHME DE BAKER

Baker a développé un algorithme de récupération adapté aux applications en temps réel s'exécutant dans un environnement à mémoire virtuelle. Cette méthode s'inspire largement de la méthode basée sur la recopie mais en la modifiant quelque peu.

La modification apportée a pour but de répartir la recopie des objets accessibles, de la zone ancienne vers la zone nouvelle, tout au long de l'exécution de l'application. Lors de chaque demande d'allocation de mémoire, un nombre fixe d'objets, disons k, sont recopiés dans la zone nouvelle. Ceci nécessite donc que les deux demi-espaces soient actifs en même temps.

En 'insérant' certaines opérations de récupération durant l'exécution de l'application, la méthode de Baker garantit un temps maximum de récupération qui soit tolérable: le temps de permuter les deux demi-espaces et de réajuster un nombre fixe de pointeurs.

Lors de chaque demande d'allocation, l'espace nécessaire à la création de l'objet sera puisé dans la zone nouvelle et un certain nombre d'objets (accesssibles) seront copiés de la zone ancienne vers la zone nouvelle. Lorsque tous les objets auront été recopiés, les deux zones pourront être permutées. L'allocation nécessitant des mots mémoire de la zone nouvelle, il est nécessaire de scinder cette dernière en deux parties distinctes: la première partie constitue la zone de création dans laquelle l'allocateur puise les mots nécessaires à la création d'un objet et la zone de recopie dans laquelle on recopie les objets accessibles de la zone ancienne.

Ceci est illustré par la figure 8.1.

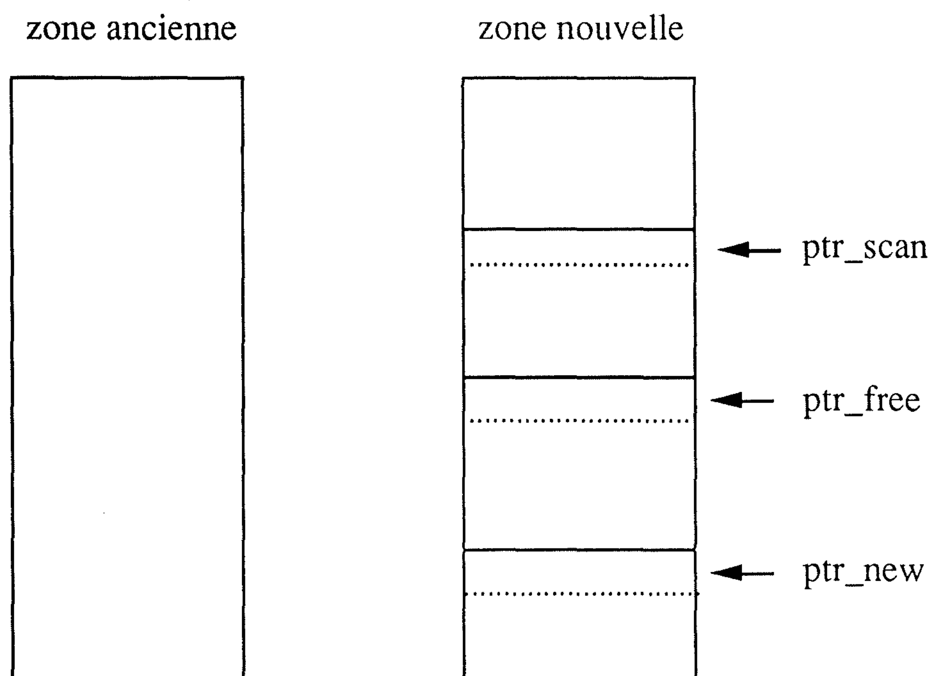


Figure 8.1

Le recopiage des objets accessibles commence par le recopiage de la racine de la structure. Ensuite, les objets recopiés sont scannés (comme dans la version normale de l'algorithme basé sur la recopie). Lorsqu'un pointeur vers la zone ancienne est trouvé, on recopie l'objet si cela n'a pas encore été réalisé et on remplace le pointeur courant de scanning par l'adresse de correspondance de l'objet recopié. Si l'objet a déjà été recopié, alors il n'y a plus qu'à remplacer le pointeur courant de scanning par l'adresse de correspondance de l'objet déjà recopié.

Le processus de scanning peut être entrelacé de demandes d'allocation. Celles-ci seront satisfaites en allouant des mots mémoire de la zone de création (dans la zone nouvelle) en commençant par allouer les mots libres d'adresses les plus grandes (à partir du bas de la mémoire).

La récupération est également gérée par deux pointeurs, **ptr_scan** et **ptr_free**. Elle se termine lorsque **ptr_scan** et **ptr_free** se rejoignent. Une fois la récupération terminée, on permute les deux zones et on recopie dans la zone nouvelle (la nouvelle zone nouvelle) tous les objets actifs de la zone ancienne (objets recopiés et objets créés).

Remarque

Après avoir créé un objet dans la partie création de la zone nouvelle, il faut être très prudent lorsqu'on y range un pointeur. En effet, il est interdit d'y mettre des pointeurs vers la zone ancienne. Avant de ranger un pointeur dans un objet de la zone de création de la zone nouvelle, il faudra toujours savoir si l'objet référencé par ce pointeur appartient à la zone ancienne ou à la zone nouvelle. S'il appartient à la zone nouvelle, alors il n'y a pas de problème. Sinon il faut recopier cet objet dans la zone nouvelle et remplacer le pointeur par son adresse de correspondance. Si l'objet a déjà été recopié, alors il suffira de remplacer le pointeur par l'adresse de correspondance de l'objet référencé.
Il en va de même pour les objets recopiés et balayés de la zone de recopie de la zone nouvelle (pour respecter l'invariant).

Cette méthode présentant de nombreuses similitudes avec la précédente, elle ne sera pas approfondie davantage.

CHAPITRE 9

LES ALGORITHMES BASES SUR LES GENERATIONS

INTRODUCTION

Dans les systèmes à allocation dynamique de la mémoire, la durée de vie des objets peut varier de façon importante. Certains objets sont utilisés pour mémoriser de l'information de façon quasi-permanente. D'autres sont utilisés par les programmes pour mémoriser de façon temporaire des résultats intermédiaires: après leur création, ces objets ne sont accessibles que durant un certain laps de temps. Passé ce délai, ils deviennent inutiles. Or, il se fait que ces objets temporaires constituent la grosse majorité des objets utilisés dans un programme. La description et l'analyse des algorithmes classiques, et même celui de Baker, nous montrent, qu'au cours de la récupération, aucune distinction n'est faite entre les objets temporaires et les objets permanents. Tous les objets, qu'ils soient temporaires ou permanents, sont récupérés de la même façon et au même moment.

L'idée qui se cache derrière la récupération par génération est de tirer profit des caractéristiques des objets suivant qu'ils sont temporaires ou permanents. En effet, l'analyse du comportement des objets nous permet de dire que dans une majorité écrasante de cas, ce sont les objets les plus jeunes qui sont le plus vite inutiles (par objet jeune, on entend un objet à qui on a alloué récemment de la mémoire). Il y a donc une forte probabilité que ce soit un objet temporaire qui devienne rapidement inaccessible au processeur. Par contre, s'il reste accessible depuis un certain temps, il y a de fortes chances pour qu'il le reste encore durant un certain temps. *L'analyse des schémas classiques d'allocation d'objets dans les programmes (en observant entre autres des phénomènes comme la création d'objets temporaires, etc) a montré qu'un objet a d'autant plus de chance de rester accessible qu'il existe depuis longtemps. C'est à dire, de manière équivalente, plus un objet est jeune, plus il a de chance de devenir du garbage.* [9]

Une bonne façon de récupérer la mémoire consisterait alors à se concentrer sur les objets les plus jeunes puisqu'ils sont les plus prometteurs du point de vue de la récupération. Ceci implique, pour le récupérateur, de pouvoir distinguer les objets temporaires des objets permanents.

L'espace adressable est divisé en plusieurs parties ou générations dont chacune contient des objets de même génération.

Le fait qu'un objet survive à plusieurs récupérations (il est toujours accessible après plusieurs récupérations), indique une certaine stabilité ou permanence de l'objet. Cet objet sera donc déplacé dans une partie de l'espace adressable correspondant à une génération directement supérieure. Lors de sa promotion, un objet passe d'une génération à une autre . Sont promus les objets qui ont survécu à un certain nombre de récupérations. Le nombre de fois qu'un objet survit au récupérateur est conservé dans un compteur qui appartient à l'objet. Le nombre de survies nécessaires au déplacement vers une partie de l'espace adressable correspondant à une génération directement supérieure peut être ajusté en fonction de la charge de travail du système.

Pour récupérer correctement une génération, tous les objets utiles, dans cette génération, doivent être accessibles. En particulier, si un objet, dans une génération, est référencé par un objet d'une autre génération, cette référence inter-générations doit être considérée comme racine et donc, il faut l'ajouter à l'ensemble des racines de la génération.

En pratique, les générations sont ordonnées par âge, et seuls les pointeurs old_to_young doivent être ajoutés à l'ensemble des racines de la génération.

Quand il devient nécessaire de récupérer une génération, cette dernière, ainsi que toutes celles qui correspondent à une génération plus jeune seront récupérées ensembles.

L'ensemble des racines du récupérateur à génération est formé:

- des registres (comme dans l'algorithme de Baker) et donc en fait des variables statiques du programme

- des pointeurs old-to-young dans la génération.

Idéalement, l'entièreté de l'espace adressable de la plus jeune génération sera toujours résidant en mémoire centrale.

La récupération par génération est utilisée par le système Smalltalk et la dernière version de la machine LISP Symbolics-3600.

Cette méthode ne sera pas davantage approfondie. En effet, une description détaillée nécessite la fixation d'un trop grand nombre de paramètres et serait donc réductrice de la réalité.

Cette méthode de récupération basée sur les générations est largement paramétrable:

- le nombre de générations. Si ce nombre est trop petit, certains objets seront promus trop tôt et les générations anciennes seront fréquemment saturées provoquant de longs garbage collection. Si ce nombre est trop grand, la promotion sera lente et il y aura un grand nombre de liens entre générations.

- choix de l'implémentation des listes de références entre générations.

- choix du type de récupération dans une génération (mark & sweep, stop & copy, Baker,...). [9]

CONCLUSION

Après avoir développé les différents algorithmes de récupération, et au vu des contraintes énoncées lors de la description de PAC, nous estimons que seuls l'algorithme basé sur le marquage et le balayage et l'algorithme basé sur la recopie sont des candidats pour être implémentés dans le système PAC.

En effet, l'algorithme basé sur le comptage des références ne récupère pas les structures cycliques. C'est un inconvénient majeur. Seules des versions 'hybrides' (marquage et balayage + comptage des références) fonctionnent mais au prix d'une lourdeur supplémentaire. De plus ils nécessitent de l'espace pour stocker les différentes tables.

L'algorithme basé sur la recopie en temps réel et l'algorithme basé sur les générations ne se justifient pas dans la mesure où ils ont été conçus pour des environnements de programmation symbolique à gros espace adressable avec mémoire virtuelle. Les machines sur lesquelles va tourner PAC ne présentent aucune de ces caractéristiques.

Partant de ces considérations, seuls l'algorithme basé sur le marquage et le balayage et l'algorithme basé sur la recopie seront abordés dans la partie Implémentation et Evaluation.

QUATRIEME PARTIE

IMPLEMENTATION ET EVALUATION

INTRODUCTION

Après avoir présenté dans la troisième partie les grandes familles de récupérateurs, nous sommes arrivés à la conclusion que seuls l'algorithme basé sur le marquage et le balayage et l'algorithme basé sur la recopie étaient des candidats pour être implémenté dans le système PAC. C'est pouquoi seuls ces deux algorithmes seront abordés dans cette quatrième partie.

Nous commencerons par définir une structure de données qui représente les objets en mémoire. La structure de données choisie devra être manipulable par les deux algorithmes de récupération de façons à rendre les tests plus commodes. Ensuite, nous donnerons une implémentation, en PASCAL, de ces algorithmes.

Nous terminerons par une évaluation basée sur les propriétés des algorithmes et sur les résultats obtenus à partir des tests.

CHAPITRE 10

IMPLEMENTATION

1. LA STRUCTURE DES DONNEES

La mémoire est un tableau de mots. Les indices de ce tableau sont compris entre **1** et la taille maximale de la mémoire soit **Taille_Mem** (0 n'est pas une adresse valide).

Un mot est un record comprenant 2 champs:

1° **id**: ce champ sert à identifier la nature du mot. Il peut valoir soit H, soit HM, soit W, soit PT, soit D.

2° **data**: ce champ contient l'information à proprement parler. Il peut contenir soit une donnée entière, soit une adresse.

Description de la nature des mots en fonction du champ Id

Id = H
Le mot dont le champ **id** est H est le mot Header de l'objet. Tout objet commence par un mot Header. Son champ **data** contient la taille totale de l'objet.

Id = HM
Le mot dont le champ **id** est HM est le mot Header d'un objet marqué lors de la phase de marquage. Son champ **data** contient la taille totale de l'objet.

Id = W
Le mot dont le champ **id** est W est un mot de travail ajouté à la représentation interne de l'objet. Son rôle est purement utilitaire.

Dans l'algorithme basé sur le marquage et le balayage, son champ **data** contient d'abord les éléments de la liste AUX puis les adresses de correspondance ("Forwarding Address") des objets marqués.

Dans l'algorithme basé sur la recopie, son champ **data** contient l'adresse de correspondance ("Forwarding Pointer") des objets accessibles.

Id = PT

Le mot dont le champ **id** est PT est un mot dont le champ **data** contient un pointeur. Comme 0 n'est pas une adresse valide, nous pouvons utiliser ce nombre pour représenter le pointeur NIL.

Id = D

Le mot dont le champ **id** est D est un mot dont le champ **data** contient une donnée.

2. LES DIFFERENTS ALGORITHMES

2.1 Algorithme à base de marquage et balayage

L'algorithme basé sur le marquage et le balayage sélectionné est celui qui compacte la mémoire de l'ordinateur lors de chaque récupération. Cette version, plus coûteuse, élimine les risques liés à l'émiettement de la mémoire.

L'implémentation de cet algorithme a été réalisée sur base des spécifications développées dans le chapitre 5, et sur base de la définition de la structure de données qui vient d'être explicitée.

On trouvera, en annexe, tout le détail du programme.

2.2 Algorithme à base de recopie

L'implémentation de cet algorithme a été réalisée sur base des spécifications développées dans le chapitre 8, et sur base de la définition de la structure de données qui vient d'être explicitée.

On trouvera, en annexe, tout le détail du programme.

CHAPITRE 11

EVALUATION et CONCLUSION GENERALE

INTRODUCTION

Nous allons, à présent, comparer les résultats des tests effectués à partir des programmes du chapitre 11 ainsi que les caractéristiques des deux méthodes.

Avant de poursuivre, il convient de bien voir que les résultats des tests ne sont pas généraux. D'autres formes d'implémentation pourraient conduire à des résultats différents.

1. EVALUATION

1.1 Complexité

Dans l'algorithme basé sur le marquage et le balayage, la phase de marquage a une complexité en O(n) où n est le nombre d'objets accessibles tandis que la phase de compactage a une complexité en O(m) où m est le nombre de mots mémoire disponibles. [5]

L'algorithme basé sur la recopie a une complexité en O(n) où n est le nombre d'objets accessibles. [5]

Dans l'algorithme basé sur la recopie, on ne recopie que les objets accessibles. Donc, plus la taille totale disponible est grande, pour un même nombre d'objets accessibles, plus le coût de récupération de l'espace inaccessible par unité de mémoire est petit. On a donc intérêt à utiliser, pour cet algorithme, un espace adressable le plus large possible. Si la taille totale disponible tend vers l'infini, le coût de récupération d'un déchet par unité de mémoire tend vers zéro. [9]

1.2 Résultats des tests

Un test a été réalisé en exécutant dix mille fois de suite la récupération d'une mémoire contenant six objets accessibles et sept objets inaccessibles. Le rapport entre le volume utile et le volume inutile est de plus ou moins 50 %. La taille de la mémoire est identique dans les deux cas. Avant chaque récupération, la mémoire est ré-initialisée. Cela représente à peu près le travail qu'un récupérateur aurait à réaliser s'il y avait dix mille racines, chacune d'elles référencant une structure équivalente (ou ayant la même forme) de six objets.

Au temps mesuré pour les dix mille cycles, il faut retirer le temps consommé pour les dix mille réinitialisations. On obtient alors le temps 'net' nécessaire aux dix mille récupérations.

Voici les résultats:

Le temps nécessaire pour exécuter dix mille récupérations basées sur le marquage et le balayage est de 96 secondes. A cela il faut soustraire le temps nécessaire aux dix mille réinitialisations qui est de 4 secondes. Nous obtenons, alors, un temps de 92 secondes pour réaliser les dix mille récupérations, ce qui nous fait 9,2 msec par récupération.

Le temps nécessaire pour effectuer dix mille récupérations basées sur la recopie s'élève à 36 secondes desquelles il faut soustraire le temps nécessaire aux dix mille ré-initialisations, soit 4 secondes. Nous obtenons, ainsi, un temps de 32 secondes pour réaliser dix mille récupérations, ce qui nous donne un délai de 3,2 msec par récupération.

Nous observons un rapport de 1 à 3 entre les performances des deux algorithmes implémentés. L'algorithme basé sur la recopie se montre le plus véloce mais au prix d'une réduction de moitié de l'espace adressable disponible. Le choix se fera en fonction de la taille de l'espace adressable disponible de l'utilisateur et de ces nécessités.

Les résultats doivent tout de même être relativisé. D'abord, si un algorithme nécessite, pour être exécuté, une zone mémoire dont la taille est comprise entre la moitié et l'entièreté de la mémoire disponible, et si le récupérateur basé sur le marquage et le balayage alors ce dernier n'aura pas à entrer en action.

Par contre, si on avait utilisé l'algorithme basé sur la recopie, on aurait dû faire appel au récupérateur puisque la mémoire disponible aurait été réduite de moitié. Enfin, l'exécution d'un algorithme nécessitant, à tout moment, plus de la moitié de la mémoire de l'ordinateur ne pourra être exécuté si la récupération utilise l'algorithme basé sur la recopie.

2. CONCLUSION GENERALE

Les deux premières parties ont montré que les systèmes de Calcul Formel nécessitaient l'emploi de machines puissantes. Les systèmes traditionnels se présentent souvent sous la forme d'un environnement fermé dont les mécanismes internes sont cachés à l'utilisateur. D'où la difficulté, pour ce dernier, de maîtriser l'exécution.

C'est pour éviter ces inconvénients que le système PAC a été développé. Ainsi, PAC fonctionne sur des machines parallèles lesquelles autorisent des performances hors du commun. De même, PAC se présentant sous la forme d'une librairie de fonctions, il laisse à l'utilisateur le contrôle total de l'exécution.

L'espace nécessaire à l'exécution des algorithmes de Calcul Formel ne cessant de croître (parfois même d'une façon exponentielle) et la taille de la mémoire étant limitée, le recours à des techniques de récupération s'est avéré fondamental.

Nous avons constaté, dans la troisième partie, que parmi les différentes familles d'algorithmes de récupération, deux d'entre elles semblaient plus appropriées pour être implémentées dans le système PAC: l'algorithme basé sur le marquage et le balayage et l'algorithme basé sur la recopie.

Finalement, dans la quatrième partie, nous avons procédé à l'implémentation et à l'évaluation de ces deux algorithmes. Nous avons constaté que, de manière générale, l'algorithme basé sur la recopie était plus rapide que l'algorithme basé sur le marquage et le balayage mais au prix d'une diminution de moitié de l'espace mémoire disponible. Ainsi, nous avons observé un rapport de un à trois entre les temps d'exécution des deux algorithmes.

Nous avons privilégié, dans ce mémoire, l'aspect clarté des spécifications et algorithmes. D'autres versions auraient peut-être permis d'accroître l'efficacité des deux algorithmes mais, selon nous, sans toutefois modifier fondamentalement le 'rapport de force' entre les deux.

BIBLIOGRAPHIE

[1] "Computer Algebra, Symbolic and Algebraic Computation";
B. Buchberger, G. E. Collins and R. Loos in cooperation with R. Albrecht;
Springer-Verlag, 1983.

[2] "Calcul formel, Systèmes et algorithmes de manipulations algébriques";
J. Davenport, Y. Siret, E. Tournier;
Masson 1987.

[3] "L'algèbre informatique";
R. Pavelle, M. Rothstein et J. Fitch;
dans "L'intelligence de l'informatique", Bibliothèque Pour La Science, 1990.

[4] "Overview of Garbage Collection in Symbolic Computing";
Timothy J. McEntee, Texas Instruments, 1987;
List Pointers I-3, 1987.

[5] "Garbage Collection of Linked Data Structures";
Jacques Cohen;
Computing Surveys, Vol. 13, N°3, September 1981.

[6] "Data Structures and Algorithms";
A. V. Aho, J. E. Hopcroft and J. D. Ullman;
Addison-Westley Publishing Company, 1983, 1987.

[7] "Calcul Formel et Parallélisme";
Jean-Louis Roch;
Laboratoire LMC-IMAG-INPG-CNRS, 1990.

[8] "Projet de Calcul Formel et Parallélisme";
J-L Roch, N. Revol, P. Sénéchaud, F. Siebert-Roch et G. Villard;
Laboratoire LMC-IMAG-INPG-CNRS, 1990.

[9] "Leçons 6 et 7 - Quelques problèmes d'implémentation";
D. Ribbens;
Chaire Francqui, Fundp 1990-1991, pp. 22-39.

[10] "List Processing in Real Time on a Serial Computer";
Henry G. Baker;
Communications of the ACM April 1978, Volume 21, Number 4,
pp 280-294.

[11] "A Real-Time Garbage Collector Based on the Lifetimes of Objetcts";
H. Lieberman and C. Hewitt;
Communications of the ACM, June 1983, Volume 26, Number 6;
pp. 419-429

[12] "Multiprocessing compactifying garbage collection";
G. L. Steele;
communication ACM 18, 9 (septembre 1975), 495-508.

[13] "A non recursive list compacting algorithm";
C. J. Cheney;
Communications of the ACM 13 (11) (1970), pp. 677-678.

[14] "Remarks on garbage collection using a two level storage";
J. Cohen and L. Trilling;
BIT 7, 1 (1967), pp. 22-30.

ANNEXES

L'ALGORITHME BASE SUR LE MARQUAGE ET LE BALAYAGE VERSION ITERATIVE

```
program Marquage_Balayage_iteratif;

const
  Taille_Mem = 120;
  Ptr_Nil = 0;

type
  pointer = integer;

  code = (H,HM,W,PT,D);

  word = record
           id: code;
           data: pointer
         end;

var
  i: integer;
  memoire: packed array [1..Taille_Mem] of word;
  free: pointer;
  racine: pointer;

{-------------------- FONCTION TAILLE -----------------------------------------}
{  TAILLE(P) est une fonction entière qui renvoie la taille (le nombre      }
{  de mots en comptant les mots d'en-tête) de l'objet pointé par le         }
{  pointeur entier P.                                                        }
{----------------------------------------------------------------------------}
function Taille (p: pointer): integer;
begin
  Taille := memoire[p].data
end;


{-------------------- FONCTION TAILLE_UTILE ----------------------------------}
{  TAILLE_UTILE(P) est une fonction entière qui renvoie la taille utile    }
{  (le nombre de mots utiles càd sans compter les mots d'en-tête) de       }
{  l'objet pointé par le pointeur entier P.                                 }
{----------------------------------------------------------------------------}
function Taille_Utile (p: pointer): integer;
begin
  Taille_Utile := memoire[p].data - 2
end;


procedure Mot (p: pointer; i: integer; var m: word);
begin
  m := memoire[p+i+1]
end;
```

```
{------------------------ FONCTION POINTEUR ------------------------------------}
{------------------------------------------------------------------------------}
function Pointeur (ptr: pointer): boolean;
begin
  if memoire[ptr].id = PT
    then Pointeur := true
    else Pointeur := false
end;


{---------------------- FONCTION NON_MARQUE ------------------------------------}
{  Non_Marque(P) est une fonction booléenne qui renvoie :                       }
{    - VRAI si l'objet pointé par le pointeur entier P n'est pas marqué,        }
{    - FAUX si l'objet pointé par le pointeur entier P est déjà marqué.         }
{------------------------------------------------------------------------------}
function Non_Marque (p: pointer): boolean;
begin
  if memoire[p].id = HM
    then Non_Marque := false
    else Non_Marque := true
end;


{---------------------- PROCEDURE MARQUER_OBJET -------------------------------}
{  MARQUER_OBJETT(P) est une procédure qui marque l'objet pointé par le         }
{  pointeur entier P en modifiant le code d'en-tête de l'objet. La             }
{  modification consiste à                                                      }
{------------------------------------------------------------------------------}
procedure Marquer_Objet (p: Pointer);
begin
  memoire[p].id := HM
end;


{------------------------ PROCEDURE MARQUER -----------------------------------}
{  MARQUER(P) est une procédure qui va marquer tous les objets accessi-         }
{  ble à partir de l'objet pointé par le pointeur entier P.                    }
{    - P pointe toujours vers le premier mot de l'objet (l'en-tête de          }
{      l'objet).                                                                }
{    - il faut marquer un objet avant de procéder au marquage de ses           }
{      "constituants". Cela empêche l'algorithme de boucler en présence        }
{      d'une structure de donnée circulaire.                                    }
{------------------------------------------------------------------------------}
procedure Marquer (p: pointer);

const
  Q0 = 1;

var
  i: integer;
  r,f: pointer;
  x: word;

begin
  Marquer_Objet(p);
  r := Q0;
  memoire[Q0+1].id := W;
  memoire[Q0+1].data := p;

  f := p;
```

```
  while (r <> f) do
    begin
      r := memoire[r+1].data;  { r <- adr(r) }

      for i := 1 to Taille_Utile(r) do
        begin
          Mot(r,i,x);
          if (x.id = PT) and (Non_Marque(x.data))
          then begin
                 Marquer_objet(x.data);
                 memoire[f+1].data := x.data;
                 f := x.data
               end
        end
    end
end;

{———————————————————— PROCEDURE FUSION ————————————————————————————}
{—————————————————————————————————————————————————————————————————}
procedure Fusion;

var
  Last: pointer;
  p: pointer;

begin
  Last := Ptr_Nil;
  p := 3;
  while (p <= Taille_Mem) and (Last = Ptr_Nil) do
    begin
      if Non_Marque(p) then Last := p;
      p := p + Taille(p);
    end;

  while p <= Taille_Mem do
    begin
      if Non_Marque(p)
      then begin
             if p = Last + Taille(Last)
             then begin
                    memoire[Last].data := memoire[Last].data + Taille(p);
                    memoire[p].id := W
                  end
             else Last := p;
           end;
      p := p + Taille(p);
    end;
  memoire[Last].data := Taille_Mem - Last + 1
end;

{——————————————————— PROCEDURE MAJ_FWD_ADD ————————————————————————}
{—————————————————————————————————————————————————————————————————}
procedure MAJ_Fwd_Add;

var
  p: pointer;
  nbre_mots_libres: integer;
```

```
begin
  p := 3;
  nbre_mots_libres := 0;
  while p <= Taille_Mem do
    begin
      if Non_marque(p)
        then nbre_mots_libres := nbre_mots_libres + Taille(p)
        else memoire[p+1].data := p - nbre_mots_libres;
      p := p + Taille(p)
    end
end;

{------------------------------ PROCEDURE MAJ_PTR ------------------------------}
{-------------------------------------------------------------------------------}
procedure MAJ_Ptr;

var
  p: pointer;
  i: integer;
  x: word;

begin
  p := 3;
  while p <= Taille_Mem do
    begin
      if not(Non_Marque(p))
        then begin
               for i := 1 to Taille_Utile(p) do
                 begin
                   Mot(p,i,x);
                   if (x.id = PT)
                   then memoire[p+i+1].data :=
                             memoire[memoire[p+i+1].data + 1].data
                 end
             end;
      p := p + Taille(p)
    end
end;

{------------------------------ PROCEDURE COPIER ------------------------------}
{-------------------------------------------------------------------------------}
procedure Copier;

var
  p: pointer;
  nbre_mots_libres: integer;
  Last: pointer;
```

```
begin
  p := 3;
  nbre_mots_libres := 0;
  while p <= Taille_Mem do
    begin
      if Non_Marque(p)
        then nbre_mots_Libres := nbre_mots_libres + Taille(p)
        else begin
               {il faut remettre le bit de marquage à 0}
               memoire[p].id := H;
               for i := p to p - 1 + Taille(p) do
                 begin
                   memoire[i - nbre_mots_libres].id := memoire[i].id;
                   memoire[i - nbre_mots_libres].data := memoire[i].data
               end;
               Last := p + Taille(p) - nbre_mots_libres
             end;
      p := p + Taille(p)
    end;

  Free := Last;
  memoire[Free].id := H;
  memoire[Free].data := Taille_Mem - Free + 1
end;


{------------------------ PROCEDURE COMPACTER ------------------------------}
{------------------------------------------------------------------------------}
procedure Compacter;
begin
  Fusion;
  MAJ_Fwd_Add;
  MAJ_Ptr;
  Copier
end;


{------------------------ PROCEDURE RECUPERER ------------------------------}
{------------------------------------------------------------------------------}
procedure Recuperer(p:pointer);
begin
  Marquer(p);
  Compacter
end;


begin
  {----- exemple -----}
  for i := 1 to Taille_Mem do
    begin
      memoire[i].id := W;
      memoire[i].data :=0
    end;
```

```
memoire[1].id := W;          memoire[1].data := 2;
memoire[2].id := W;          memoire[2].data := 3;

memoire[3].id := H;          memoire[3].data := 5;
memoire[4].id := W;          memoire[4].data := 999;
memoire[5].id := PT;         memoire[5].data := 13;
memoire[6].id := PT;         memoire[6].data := 22;
memoire[7].id := PT;         memoire[7].data := 31;

memoire[8].id := H;          memoire[8].data := 5;
memoire[9].id := W;          memoire[9].data := 999;
memoire[10].id := D;         memoire[10].data := 0;
memoire[11].id := D;         memoire[11].data := 0;
memoire[12].id := D;         memoire[12].data := 0;

memoire[13].id := H;         memoire[13].data := 4;
memoire[14].id := W;         memoire[14].data := 999;
memoire[15].id := PT;        memoire[15].data := 40;
memoire[16].id := D;         memoire[16].data := 0;

memoire[17].id := H;         memoire[17].data := 5;
memoire[18].id := W;         memoire[18].data := 999;
memoire[19].id := D;         memoire[19].data := 0;
memoire[20].id := D;         memoire[20].data := 0;
memoire[21].id := D;         memoire[21].data := 0;

memoire[22].id := H;         memoire[22].data := 4;
memoire[23].id := W;         memoire[23].data := 999;
memoire[24].id := PT;        memoire[24].data := 48;
memoire[25].id := D;         memoire[25].data := 0;

memoire[26].id := H;         memoire[26].data := 5;
memoire[27].id := W;         memoire[27].data := 999;
memoire[28].id := D;         memoire[28].data := 0;
memoire[29].id := D;         memoire[29].data := 0;
memoire[30].id := D;         memoire[30].data := 0;

memoire[31].id := H;         memoire[31].data := 4;
memoire[32].id := W;         memoire[32].data := 999;
memoire[33].id := D;         memoire[33].data := 0;
memoire[34].id := D;         memoire[34].data := 0;

memoire[35].id := H;         memoire[35].data := 5;
memoire[36].id := W;         memoire[36].data := 999;
memoire[37].id := D;         memoire[37].data := 0;
memoire[38].id := D;         memoire[38].data := 0;
memoire[39].id := D;         memoire[39].data := 0;

memoire[40].id := H;         memoire[40].data := 3;
memoire[41].id := W;         memoire[41].data := 999;
memoire[42].id := D;         memoire[42].data := 0;

memoire[43].id := H;         memoire[43].data := 5;
memoire[44].id := W;         memoire[44].data := 999;
memoire[45].id := D;         memoire[45].data := 0;
memoire[46].id := D;         memoire[46].data := 0;
memoire[47].id := D;         memoire[47].data := 0;

memoire[48].id := H;         memoire[48].data := 3;
memoire[49].id := W;         memoire[49].data := 999;
memoire[50].id := D;         memoire[50].data := 0;
```

```
  memoire[51].id := H;        memoire[51].data := 5;
  memoire[52].id := W;        memoire[52].data := 999;
  memoire[53].id := D;        memoire[53].data := 0;
  memoire[54].id := D;        memoire[54].data := 0;
  memoire[55].id := D;        memoire[55].data := 0;

  memoire[56].id := H;        memoire[56].data := 5;
  memoire[57].id := W;        memoire[57].data := 999;
  memoire[58].id := D;        memoire[58].data := 0;
  memoire[59].id := D;        memoire[59].data := 0;
  memoire[60].id := D;        memoire[60].data := 0;
  memoire[61].id := H;        memoire[61].data := 60;

  {-----------------}

  racine := 3;

  recuperer(racine);

  for i := 1 to Free do
    begin
      if memoire[i].id = H   then writeln(i,' H  ',memoire[i].data);
      if memoire[i].id = HM  then writeln(i,' HM ',memoire[i].data);
      if memoire[i].id = W   then writeln(i,' W  ',memoire[i].data);
      if memoire[i].id = PT  then writeln(i,' PT ',memoire[i].data);
      if memoire[i].id = D   then writeln(i,' D  ',memoire[i].data);
      readln
    end;

  writeln('Free=', free);
  readln
end.
```

L'ALGORITHME BASE SUR LA RECOPIE
VERSION ITERATIVE

```
program Recopie_iteratif;

const
  Taille_Mem = 120;
  Ptr_Nil = 0;

type
  pointer = integer;

  code = (H,HM,W,PT,D);

  word = record
           id: code;
           data: pointer
         end;

var
  i: integer;
  memoire: packed array [1..Taille_Mem] of word;
  free: pointer;
  racine: pointer;

{------------------- FONCTION TAILLE -------------------------------------------}
{  TAILLE(P) est une fonction entière qui renvoie la taille (le nombre         }
{  de mots en comptant les mots d'en-tête) de l'objet pointé par le            }
{  pointeur entier P.                                                          }
{------------------------------------------------------------------------------}
function Taille (p: pointer): integer;
begin
  Taille := memoire[p].data
end;


{------------------- FONCTION TAILLE_UTILE -------------------------------------}
{  TAILLE_UTILE(P) est une fonction entière qui renvoie la taille utile        }
{  (le nombre de mots utiles càd sans compter les mots d'en-tête) de           }
{  l'objet pointé par le pointeur entier P.                                    }
{------------------------------------------------------------------------------}
function Taille_Utile (p: pointer): integer;
begin
  Taille_Utile := memoire[p].data - 2
end;


{--------------------- FONCTION FWD_PTR ----------------------------------------}
{------------------------------------------------------------------------------}
function Fwd_Ptr(p:pointer):pointer;
begin
  Fwd_Ptr := memoire[p+1].data
end;
```

```
{------------------- FONCTION MOT --------------------------------------------}
{  MOT(P,I) est une fonction entière qui renvoie le i ème mot utile de      }
{  l'objet qui est pointé par le pointeur entier P.                           }
{-----------------------------------------------------------------------------}
function Mot (p: pointer; i: integer): integer;
begin
  mot := memoire[p+i+1].data
end;


{------------------- FONCTION NEWSPACE ---------------------------------------}
{  NEWSPACE(P) est une fonction booléenne qui renvoie :                        }
{    - VRAI si l'objet pointé par le pointeur entier P est ds le newspace }
{    - FAUX si l'objet pointé par le pointeur entier P n'est pas  "          }
{-----------------------------------------------------------------------------}
function Newspace (p: pointer): boolean;
begin
  if p >= ((Taille_Mem div 2) + 1)
    then Newspace := true
    else Newspace := false
end;


{------------------- FONCTION OLDSPACE ---------------------------------------}
{  OLDSPACE(P) est une fonction booléenne qui renvoie :                        }
{    - VRAI si l'objet pointé par le pointeur entier P est ds le oldspace }
{    - FAUX si l'objet pointé par le pointeur entier P n'est pas  "          }
{-----------------------------------------------------------------------------}
function Oldspace (p: pointer): boolean;
begin
  if (p >= 1) and (p <= Taille_Mem div 2)
    then Oldspace := true
    else Oldspace := false
end;


{-------------------- FONCTION POINTEUR --------------------------------------}
{-----------------------------------------------------------------------------}
function Pointeur(p: pointer): boolean;
begin
  if memoire[p].id = PT
    then Pointeur := true
    else Pointeur := false
end;


{----------------------- PROCEDURE RECOPIE -----------------------------------}
{                                                                             }
{-----------------------------------------------------------------------------}
procedure recopie(p: pointer);
var
  i: integer;
  ptr_scan: pointer;
  ptr_free: pointer;
```

```
begin

  { initialisation }

  ptr_free := (taille_Mem div 2) +1; (1ère adresse de la zone nouvelle)
  ptr_scan := (taille_Mem div 2) +1;

  { ————————— recopie de l'objet racine (pointé par p) ————————— }

  if Newspace(p) { p appartient à newspace }
  then { ne rien faire }
  else begin
           { on ne vérifie pas pas le fwd_ptr(p) pcq, comme on commence la }
           { récupération, on est sur qu'on ne l'a pas encore recopié.      }
           { recopie de l'objet }

           for i := 1 to Taille(p) do
             begin
               memoire[ptr_free+i-1].id := memoire[p+i-1].id;
               memoire[ptr_free+i-1].data := memoire[p+i-1].data;
             end;

           { à ce moment, l'objet racine a été recopié et il faut encore }
           { modifier le forwarding pointer de l'objet d'adresse p dans  }
           { la zone ancienne.                                           }

           memoire[p+1].data := ptr_free;

           { mise à jour du prt_free }

           ptr_free := ptr_free + Taille(p);

       end;

  { Fin de l'initialisation }

  { Début itération }
  while ptr_free <> ptr_scan do
    begin
      if pointeur(ptr_scan) { si p est un pointeur }
      then begin
              { Il ne faut pas tester si p pointe vers la nouvelle zone, cas }
              { dans lequel il ne faudrait rien faire, pcq les pointeurs des }
              { objets de la zone nouvelle ne peuvent avoir été modifiés     }
              { avant d'être scannés. Les pointeurs sont donc ceux qu'il y   }
              { avait dans la zone ancienne.                                 }

              p := memoire[ptr_scan].data;

              if fwd_ptr(p) = Ptr_Nil
              then begin
                      { l'objet p n'a pas encore été copié et il faut le faire }
                      for i := 1 to Taille(p) do
                        begin
                          memoire[ptr_free+i-1].id := memoire[p+i-1].id;
                          memoire[ptr_free+i-1].data := memoire[p+i-1].data;
                        end;

                      { l'objet est copié }
```

```
                  { mise à jour du forwarding pointer de l'objet qui }
                  { vient d'être copié                               }

                  memoire[p+1].data := ptr_free;

                 {modification du mot pointé par ptr_scan (qui contenait p)}

                  memoire[ptr_scan].data := ptr_free;

                  { mise à jour de ptr_free }

                  ptr_free := ptr_free + Taille(p)
                end
             else begin
                  { l'objet a déjà été copié et son adresse est }
                  { fwd_ptr(p)                                  }
                  memoire[ptr_scan].data := Fwd_Ptr(p)
                end
           end;

      ptr_scan := ptr_scan + 1
     end;
  free := ptr_free
end;


begin

  for i := 1 to Taille_Mem do
    begin
      memoire[i].id := W;
      memoire[i].data :=0
    end;

  memoire[1].id := W;          memoire[1].data := 2;
  memoire[2].id := W;          memoire[2].data := 3;

  memoire[3].id := H;          memoire[3].data := 5;
  memoire[4].id := W;          memoire[4].data := Ptr_Nil;
  memoire[5].id := PT;         memoire[5].data := 13;
  memoire[6].id := PT;         memoire[6].data := 22;
  memoire[7].id := PT;         memoire[7].data := 31;

  memoire[8].id := H;          memoire[8].data := 5;
  memoire[9].id := W;          memoire[9].data := Ptr_Nil;
  memoire[10].id := D;         memoire[10].data := 0;
  memoire[11].id := D;         memoire[11].data := 0;
  memoire[12].id := D;         memoire[12].data := 0;

  memoire[13].id := H;         memoire[13].data := 4;
  memoire[14].id := W;         memoire[14].data := Ptr_Nil;
  memoire[15].id := PT;        memoire[15].data := 40;
  memoire[16].id := D;         memoire[16].data := 0;

  memoire[17].id := H;         memoire[17].data := 5;
  memoire[18].id := W;         memoire[18].data := Ptr_Nil;
  memoire[19].id := D;         memoire[19].data := 0;
  memoire[20].id := D;         memoire[20].data := 0;
  memoire[21].id := D;         memoire[21].data := 0;
```

```
memoire[22].id := H;        memoire[22].data := 4;
memoire[23].id := W;        memoire[23].data := Ptr_Nil;
memoire[24].id := PT;       memoire[24].data := 48;
memoire[25].id := D;        memoire[25].data := 0;

memoire[26].id := H;        memoire[26].data := 5;
memoire[27].id := W;        memoire[27].data := Ptr_Nil;
memoire[28].id := D;        memoire[28].data := 0;
memoire[29].id := D;        memoire[29].data := 0;
memoire[30].id := D;        memoire[30].data := 0;

memoire[31].id := H;        memoire[31].data := 4;
memoire[32].id := W;        memoire[32].data := Ptr_Nil;
memoire[33].id := D;        memoire[33].data := 0;
memoire[34].id := D;        memoire[34].data := 0;

memoire[35].id := H;        memoire[35].data := 5;
memoire[36].id := W;        memoire[36].data := Ptr_Nil;
memoire[37].id := D;        memoire[37].data := 0;
memoire[38].id := D;        memoire[38].data := 0;
memoire[39].id := D;        memoire[39].data := 0;

memoire[40].id := H;        memoire[40].data := 3;
memoire[41].id := W;        memoire[41].data := Ptr_Nil;
memoire[42].id := D;        memoire[42].data := 0;

memoire[43].id := H;        memoire[43].data := 5;
memoire[44].id := W;        memoire[44].data := Ptr_Nil;
memoire[45].id := D;        memoire[45].data := 0;
memoire[46].id := D;        memoire[46].data := 0;
memoire[47].id := D;        memoire[47].data := 0;

memoire[48].id := H;        memoire[48].data := 3;
memoire[49].id := W;        memoire[49].data := Ptr_Nil;
memoire[50].id := D;        memoire[50].data := 0;

memoire[51].id := H;        memoire[51].data := 5;
memoire[52].id := W;        memoire[52].data := Ptr_Nil;
memoire[53].id := D;        memoire[53].data := 0;
memoire[54].id := D;        memoire[54].data := 0;
memoire[55].id := D;        memoire[55].data := 0;

memoire[56].id := H;        memoire[56].data := 5;
memoire[57].id := W;        memoire[57].data := Ptr_Nil;
memoire[58].id := D;        memoire[58].data := 0;
memoire[59].id := D;        memoire[59].data := 0;
memoire[60].id := D;        memoire[60].data := 0;

{----------------}

racine := 3;

recopie(racine);

for i := 1 to Free do
  begin
    if memoire[i].id = H   then writeln(i,' H  ',memoire[i].data);
    if memoire[i].id = HM  then writeln(i,' HM ',memoire[i].data);
    if memoire[i].id = W   then writeln(i,' W  ',memoire[i].data);
    if memoire[i].id = PT  then writeln(i,' PT ',memoire[i].data);
    if memoire[i].id = D   then writeln(i,' D  ',memoire[i].data);
    readln
  end;
```

```
  writeln('free=', free);
  readln

end.
```

Réalisation, impression et reliure : SPRL Centre de Reprographie - Tél. : 081/22.87.72