Réalisation d'un Glaneur de Cellules de Lang et Dupont à générations

Damien Doligez

Mémoire de D.E.A., Université de Paris VII;
sous la direction de Jean-Jacques Lévy

lundi 18 septembre 1989

Réalisation d'un Glaneur de Cellules de Lang et Dupont à générations

/* […] Note that this could cause
* a memory fault if the old area was tiny, and the moon
* is gibbous. However, that is very unlikely.
*/

Chris Kingsley, *malloc.c,*
sources de UNIX BSD

Résumé

Le Glaneur de Cellules (*Garbage Collector*) est aujourd'hui un composant essentiel dans l'implémentation de tous les langages de haut niveau. Il existe une grande variété d'environnements différents, chacun avec ses contraintes particulières; et un bon nombre d'algorithmes ont été développés pour répondre à ces contraintes : comptes de références (*reference counting*), balayage (*mark and sweep*), copie (*stop and copy*), leurs versions incrémentales, et enfin les systèmes à générations. Après un rapide survol des techniques existantes, nous verrons les raisons qui m'ont fait choisir l'algorithme de Lang et Dupont (qui mélange balayage et copie), et le détail de l'implémentation (structures de données et algorithme) ainsi que toutes les fonctionnalités annexes qui font la différence entre un GC et un gestionnaire de mémoire (allocation, contrôle du GC…) Enfin, on présentera quelques mesures de performances.

Sommaire

1. Introduction. 1
2.1. Algorithmes de base. 2
2.2. Générations. 8
2.3. Algorithmes incrémentaux et parallèles. 10
2.4. Algorithme de Lang et Dupont. 11
2.5. Comparaison des différents algorithmes.. 12
3. Implémentation. 13
3.1. Choix stratégiques. 13
3.2. Le modèle mémoire. 13
3.3. Implémentation de l'algorithme de Lang et Dupont. 14
3.4. Le mécanisme de générations. 18
4. Performances. 20
5. Conclusion. 23

1. Introduction

Dans les langages modernes de haut niveau, tels que LISP, Smalltalk, ML, …, on manipule des objets symboliques : ce sont des arbres, des listes, des graphes, dont les flèches sont représentées par des pointeurs, et dont les nœuds sont des objets *alloués* dans la mémoire, c'est-à dire des emplacements de la mémoire qui sont réservés au moment de la création des nœuds en question. Lorsque les nœuds ne sont plus utiles, il faut les *désallouer*, pour récupérer la mémoire qu'ils occupent.

Un objet est alloué explicitement (par les primitives `cons`, `new`, …) par le programme, mais celui-ci n'a aucun contrôle sur la désallocation des objets : il n'existe pas de primitive pour désallouer (il y en a une en LISP, mais son emploi est marginal.) Une telle primitive ne serait pas sûre, c'est-à dire que son comportement pourrait être aléatoire dans le cas où on désallouerait un objet référencé par plusieurs pointeurs. En d'autres termes, sa sémantique ne serait pas complètement définie.

Comme il n'y a pas de désallocation explicite, c'est le système qui doit identifier les objets qui sont devenus inutiles et les désallouer. Cette tâche est connue sous le nom de *glanage de cellules* (en anglais : *garbage collection*), que l'on abrège par GC. Ce mécanisme introduit une perte de performances qui est largement compensée par la sécurité qu'il apporte (c'est une source d'erreurs qui disparaît.) De plus, à partir d'une certaine complexité, la gestion des allocations et désallocations par le programmeur devient impossible en pratique.

Depuis l'introduction de LISP en 1958, un grand nombre d'algorithmes ont été imaginés, avec des variantes adaptées à des contextes très variés. Pour une étude plus complète de cette variété d'algorithmes, on se reportera aux travaux de Jacques Cohen [Coh 81], et d'Eric Spir [Spi 89].

La grande variété d'algorithmes de GC existants s'explique par les contextes très variés d'utilisation.

Dans les algorithmes non incrémentaux, le collecteur (le programme de gestion de la mémoire) interrompt le mutateur[1] (le programme qui effectue les calculs) pour effectuer un cycle de GC. La durée de cette interruption est appellée *temps de latence*. Suivant l'utilisation du système, on a des contraintes sur le temps de latence qui peuvent aller de "aucune contrainte" à "quelques millisecondes au maximum".

Le GC a un coût en temps, qui est la portion du temps de calcul total consacrée au GC, et un coût en mémoire, qui est la proportion entre la place occupée par les objets vivants et la taille totale de la mémoire. Il peut être plus ou moins important de minimiser l'un ou l'autre de ces coûts, selon le contexte.

Il peut être important de tenir compte de la mémoire virtuelle : ses performances sont meilleures si la localité des références est assez grande, c'est-à dire si les objets pointent sur des objets qui sont proches d'eux dans la mémoire.

La taille moyenne des objets manipulés constitue aussi un paramètre important : certains systèmes ont des objets assez gros (plusieurs dizaines de pointeurs en moyenne), tandis que d'autres (par exemple Lisp et ML) utilisent des objets plus petits, contenant deux ou trois pointeurs.

La diversité des contraintes motive l'existence d'un grand nombre d'algorithmes de GC, et il est très important de choisir un algorithme adapté au système considéré. J'ai donc réalisé la première implémentation de l'algorithme de Lang et Dupont avec des générations, en vue d'une utilisation en vraie grandeur dans le système CAML, dont le GC actuel est remarquablement inefficace.

Après un bref rappel des algorithmes de GC existants, je vais détailler l'implémentation du GC de Lang et Dupont à générations. Je présente ensuite quelques mesures de performances, comparées à celles des algorithmes classiques à copie et à balayage.

1. Les termes *collecteur* et *mutateur* ont été introduits par Dijkstra dans [Dij 78].

2. Algorithmes

Je vais détailler les algorithmes de GC existants : d'abord les trois algorithmes de base, puis le principe des générations, la façon de paralléliser les algorithmes, l'algorithme de Lang et Dupont. Je présente ensuite un tableau récapitulatif.

2.1. Algorithmes de base

On va étudier les trois algorithmes de base : comptes de références, balayage, et copie, en s'intéressant au coût en temps et en mémoire et a la possibilité de parallélisation.

2.1.1. Comptes de références

Principe

Les GC à compte de références (*reference count*) reposent sur le principe suivant :

Comme on cherche à savoir quels sont les objets qui ne sont pas pointés, il suffit de tenir à jour un compte du nombre de pointeurs pointant chaque objet. A chaque fois qu'on crée un pointeur vers cet objet, on ajoute un au compte, et quand on détruit un pointeur, on retire un. Lorsque le compte tombe à zéro, on sait que l'objet est inactif.

On peut créer un pointeur vers un objet :

• en allouant l'objet. On crée alors l'objet avec un compte égal à 1.

• en copiant un pointeur vers un objet existant. On incrémente alors le compteur de cet objet.

On peut détruire un pointeur vers un objet :

• en affectant une autre valeur à la variable où ce pointeur se trouvait.

• en libérant un objet dont le compte est tombé à zéro et qui contenait ce pointeur.

Avantages

Le principal avantage des GC à compte de références est qu'ils sont naturellement incrémentaux, alors que les implémentations[2] naïves des autres méthodes introduisent des temps de latence qui peuvent être importants (2 à 3 secondes en moyenne pour CAML [FOR 89; C&H 87] avec un GC à copie.)

Défauts

Les principaux défauts de cette méthode sont au nombre de deux :

• Les structures circulaires ne sont pas récupérées : elles deviennent inactives sans que le compte de leurs objets tombe à zéro (voir figure 1.)

• Le coût en temps du GC à compte de référence est très élevé, puisqu'il faut mettre à jour un champ de l'objet à chaque affectation de pointeur (voir figure 2.) En fait, il est proportionnel au nombre de manipulations de pointeurs effectuées par le mutateur.

Notons aussi qu'un GC à compte de références occupe beaucoup de place dans le code compilé du mutateur, puisque les manipulations de pointeurs deviennent beaucoup plus compliquées.

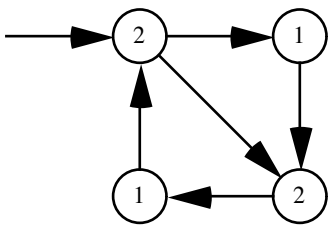
Exemple

Examinons le code du GC de Smalltalk tel qu'on le trouve dans [G&R 83] (voir encadré 1)

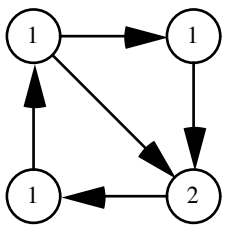
A chaque fois que le mutateur veut accéder à un pointeur ou écrire une valeur dans un champ d'un objet, il appelle le gestionnaire de mémoire qui effectue les mises à jour. C'est bien sûr extrêmement coûteux. On a un coût moindre s'il s'agit d'un langage compilé, puisqu'on peut écrire en assembleur les manipulations des comptes de références, pour éviter le coût d'un appel de procédure (c'est la technique utilisée dans [DeT 88].)

De plus, il se pose un problème à la désallocation : lorsque le compte d'un objet tombe à zéro, il faut le

2. J'utilise le terme anglais *implémentation* en tant que néologisme, que je trouve plus élégant que *implantation*.



avant



après

Les chiffres indiqués dans les objets sont les comptes de références.

figure 1 : libération d'un pointeur sur une structure circulaire.

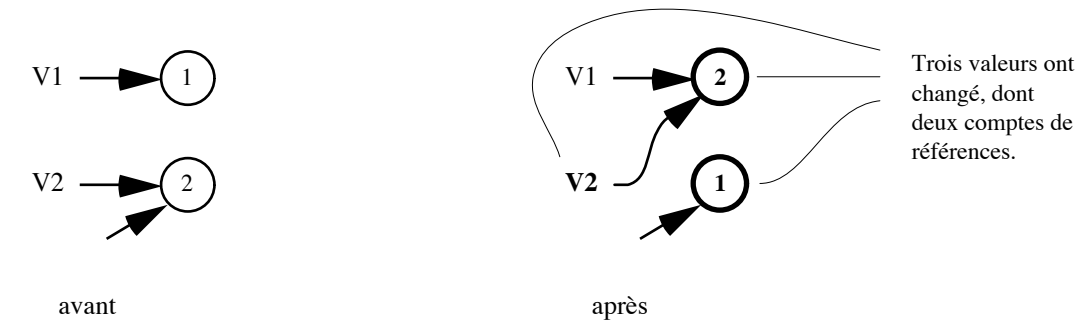


figure 2 : accès mémoire dûs à une affectation de pointeur.

libérer, et ce faisant on détruit tous les pointeurs qu'il contient. Si l'un des objets pointés par ces pointeurs doit être libéré à son tour, il faut faire un appel récursif à la routine de désallocation. Ce phénomène peut conduire à un débordement de pile. Pour l'éviter, on peut recourir à la technique de l'inversion de pointeurs (voir figure 3.) Il se trouve aussi que ce parcours du sous-graphe à désallouer peut prendre un temps arbitrairement long, ce qui introduit un temps de latence gênant pour une application temps-réel (ce problème est moins important pour un système interactif, puisque le temps de latence moyen est en revanche très court.)

Variantes

Les GC à compte de référence ont été utilisés notamment pour Smalltalk [G&R 83] et Modula 2+ [DeT 88]. Cependant, il est intéressant de signaler que, dans un cas comme dans l'autre, on les a associés à un GC auxiliaire (à balayage ou à copie) pour récupérer les structures circulaires : on fait fonctionner le système avec le GC à compte de références, et lorsque la mémoire est remplie de structures circulaires inaccessibles, on déclenche le GC auxiliaire. Cette technique permet de résoudre le problème des structures circulaires, au prix d'un système plus compliqué à mettre en œuvre, puisqu'il faut écrire deux GC.

On peut gagner de la place en utilisant un compte très petit : un seul bit suffit à récupérer une grande majorité des objets[3]. Les autres sont sacrifiés, ou bien jetés en pâture au GC auxiliaire. De plus, on gagne du temps en utilisant une arithmétique simplifiée (voir encadré 2.)

Le coût en temps pose un problème plus important, en particulier à cause de la localité des références. En effet, lorsqu'on manipule un objet, on est obligé d'accéder à un champ de cet objet, ce qui se révèle très coûteux, surtout dans un contexte de mémoire virtuelle : le GC introduit énormément d'accès mémoire parasites qui ne sont pas forcément proches des accès réalisés par le mutateur.

On peut y remédier en différant les opérations de mise à jour des comptes de références. En effet, on peut alors simplifier certaines opérations qui s'annulent. Cette technique permet notamment d'éviter de mettre à jour les comptes des objets temporaires, qui sont précisément très nombreux en ML.

On peut éviter le temps de latence dû à la désallocation en chaîne en appliquant la technique suivante : on diffère la désallocation en plaçant l'objet à désallouer dans une liste spéciale, ou bien on peut réalise cette opération en parallèle. On peut aussi faire la mise à jour des comptes en parallèle avec le mutateur si on a une machine multi-processeurs, comme dans [DeT 88].

2.1.2. Balayage

Principe

Dans un GC à balayage (*mark and sweep*) on commence par arrêter le mutateur, puis on effectue un parcours complet du graphe des objets en mémoire, en coloriant les objets parcourus. Quand ce parcours est fini, on sait alors que les objets coloriés sont actifs, et que tous les autres sont inactifs. La deuxième phase parcourt linéairement la mémoire et désalloue les objets qui ne sont pas coloriés.

Avantages

Les GC à balayage sont assez faciles à implémenter incrémentalement ou en parallèle, surtout par rapport aux GC à copie.

Ils sont souvent moins coûteux en temps que les GC à compte de références.

Enfin, ils demandent peu de place en mémoire, puisqu'il suffit d'un ou deux bits de couleur par objet.

3. La proportion exacte dépend évidemment du langage et du programme étudiés.

encadré 1 : le GC de Smalltalk (en Smalltalk)

```
countUp: objectPointer
    | count |
    (self isIntegerObject: objectPointer)
        ifFalse:
            [count ← (self countBitsOf: objectPointer) + 1.
             count < 129 ifTrue: [self countBitsOf: objectPointer put: count]].
    ↑objectPointer

countDown: rootObjectPointer
    | count |
    (self isIntegerObject: rootObjectPointer)
        ifTrue: [↑rootObjectPointer]
        ifFalse: "this is a pointer, so decrement its reference count"
            [↑self forAllObjectsAccessibleFrom: rootObjectPointer
                suchThat:
                    "the predicate decrements the count and tests for zero"
                    [:objectPointer |
                     count ← (self countBitsOf: objectPointer) - 1.
                     count < 127
                        ifTrue: [self countBitsOf: objectPointer put: count].
                     count = 0]
                do: "the action zeroes the count and deallocates the object"
                    [:objectPointer |
                     self countBitsOf: objectPointer put: 0.
                     self deallocate: object Pointer]]

forAllObjectsAccessibleFrom: objectPointer
        suchThat: predicate
        do: action
    (predicate value: objectPointer)
        ifTrue:
            [↑self forAllOtherObjectsAccessibleFrom: objectPointer
                suchThat: predicate
                do: action]

forAllOtherObjectsAccessibleFrom: objectPointer
        suchThat: predicate
        do: action
    | next |
    1 to: (self lastPointerOf: objectPointer) - 1 do:
        [:offset |
         next ← self heapChunkOf: objectPointer word: offset.
         ((self isIntegerObject: next)==false and: [predicate value: next])
            ifTrue: "it's a non-immediate object and it should be processed"
                [self forAllOtherObjectsAccessibleFrom: next
                    suchThat: predicate
                    do: action]].
    "all pointers have been followed; now perform the action"
    action value: objectPointer.
    ↑objectPointer
```

Défauts

Le coût en temps du GC à balayage est plus élevé que celui du GC à copie, et lors du parcours du graphe, la hauteur de pile peut devenir très importante. En effet, si on effectue un parcours arborescent récursif, la profondeur de l'arbre de parcours n'est bornée que par le nombre de nœuds parcourus, c'est-à dire par le nombre d'objets actifs; et cette profondeur est aussi la hauteur de la pile nécessaire au parcours.

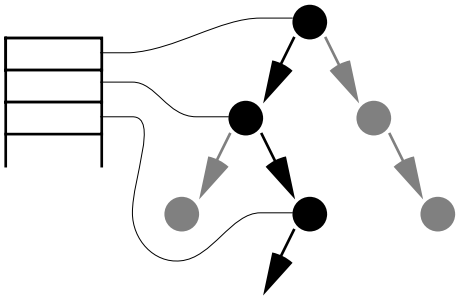
Exemple

On peut examiner (encadré 3) le texte du GC proposé par Ben-Ari dans [Ben 84]. Ici, le problème de la pile est résolu au prix d'un énorme surcoût en temps de calcul, puisqu'on balaye plusieurs fois toute la mémoire pour colorier les objets accessibles, au lieu de faire un parcours simple du graphe des objets accessibles.

Variantes

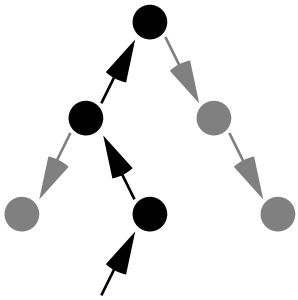
Les GC à balayage ont été utilisés dans de nombreux systèmes LISP; en particulier LE_LISP [Cha 86]. Ils ont aussi fait l'objet d'études théoriques

Parcours d'arbre classique avec une pile.



La pile mémorise la branche en cours de traitement. Pour remonter d'un niveau, on dépile.

Parcours avec inversion de pointeurs.



Les pointeurs inverses mémorisent l'information qui était dans la pile. Pour remonter d'un niveau, on suit le pointeur inversé et on le remet à l'endroit.

figure 3 : l'inversion de pointeurs

diverses : comme exercice de preuve de correction [Dij 78; K&S 77b; Ben 84] ou du point de vue de l'efficacité [K&S 77a].

Les techniques utilisées pour le parcours de graphe sans utiliser de pile sont au nombre de deux : utilisation d'une troisième couleur (envisagée dans [Dij 78] et [L&D 88]) ou inversion de pointeurs. Celle-ci exige pas mal de manipulations de la mémoire, et elle est relativement compliquée à mettre en œuvre (voir figure 3.) L'utilisation de la troisième couleur est détaillée dans l'encadré 4.

On peut noter aussi une variante intéressante proposée dans [QBQ 89], qui effectue le balayage d'un cycle de GC pendant le coloriage du cycle suivant.

2.1.3. Copie

Principe

Cet algorithme (*Stop and copy* ou *copying GC*) est sans doute le moins naturel des trois, puisqu'il exige de recopier tous les objets actifs. La mémoire est découpée en deux zones. L'une d'elles (l'espace de destination, ou *to-space*) est réservée au GC et elle reste vide pendant que le mutateur tourne, et alloue dans l'autre zone (espace de départ, ou *from-space*) jusqu'à ce qu'elle soit pleine. On arrête alors le mutateur et on parcourt le graphe des objets actifs en copiant chaque objet actif dans l'espace d'arrivée (voir figure 4.) Il faut bien sûr mettre à jour tous les pointeurs puisqu'on déplace les objets.

Le parcours du graphe se fait en largeur, et l'espace d'arrivée sert de FIFO pour ce parcours. L'espace d'arrivée est donc découpé en trois parties : les objets qu'on a copié et dont on a copié les fils (les objets qui sont sortis de la FIFO) les objets qu'on a copiés et dont on n'a pas copié les fils (les objets qui sont dans la FIFO) et l'espace disponible (où on place les objets qui rentrent dans la FIFO). On a deux pointeurs qui séparent ces trois zones : le pointeur de copie, et le pointeur de balayage (les deux bornes de la FIFO.)

Avantages

Paradoxalement, cette méthode est très rapide. En effet, le coût en temps du GC est seulement proportionnel au nombre d'objets actifs au moment

encadré 2 : Arithmétique de {1, ∞}

incrémentation :
$1 \rightarrow \infty$
$\infty \rightarrow \infty$

décrémentation :
$1 \rightarrow 0$ (on sort du domaine, il faut désallouer le bloc)
$\infty \rightarrow \infty$

on a donc le pseudo-code suivant :

```
procedure incrementer_compte (x)
begin
    if compte (x) = 1
    then compte (x) := infini
end

procedure decrementer_compte (x)
begin
    if compte (x) = 1
    then desallouer (x)
end
```

Il suffit donc d'un test et une affectation. Aucune addition ni soustraction n'est nécéssaire.

encadré 3 : l'algorithme de Ben-Ari (en Ada)

```
-- The collector
    -- The collector executes procedure Marking_Phase and then
    -- appends white nodes to the free list.
    procedure Marking_phase is
        Black_Count, Old_Black_Count: Integer := 0;
    begin
        -- Blacken the roots.
            for I in Roots loop
                M(I).Color := Black;
            end loop;
        Main: loop
        -- Propagate the coloring.
            for I in Index loop
                if M(I).Color = Black then
                    for J in Sons loop
                        M( M(i).Sons(J) ).Color := Black;
                    end loop;
                end if;
            end loop;
        -- Count the number of Black nodes.
            Black_Count := 0;
            for I in Index loop
                if M(I).Color = Black then
                    Black_Count := Black_count + 1;
                end if;
            end loop;
        -- Repeat the Main loop if more Black nodes than before.
            if Black_Count > Old_Black_Count
                then Old_Black_Count := Black_Count;
                else exit;
            end if;
        end loop Main;
    end Marking_Phase;

        -- Collector Cycle.
        Marking_Phase;
        -- Collect the white nodes.
            for I in Index loop
                if M(I).Color = White
                    then Append_to_Free(I);
                    else M(I).Color := White;
                end if;
            end loop;
```

où le GC se déclenche.

Le deuxième avantage est le compactage de la mémoire réalisé par ce GC. En effet, à la fin du GC, on récupère toute la mémoire libre sous forme d'un seul bloc contigu; ce qui permet d'avoir une méthode d'allocation très rapide (pour une explication détaillée des problèmes d'allocation et de fragmentation, on se reportera à l'œuvre de Knuth [Knu 68].)

Enfin, on peut facilement écrire une structure de données sur le disque en déclenchant un GC qui regroupe tous les objets en mémoire, et en sauvant la portion de mémoire où ils se trouvent (ou bien en sauvant les objets sur le disque au fur et à mesure, au lieu de les copier dans l'espace d'arrivée.) Cette technique est utilisée dans Modula 2+, Amber [Car 86], SML, …

Défauts

Le défaut évident de cette méthode est qu'elle réserve la moitié de la mémoire pour les besoins du GC.

Un autre problème est que cette méthode est très difficile à paralléliser ou même à incrémentaliser, puisqu'elle déplace les objets (il faut mettre à jour tous les pointeurs sur un objet lorsqu'on déplace cet objet, avant que le mutateur utilise ces pointeurs.)

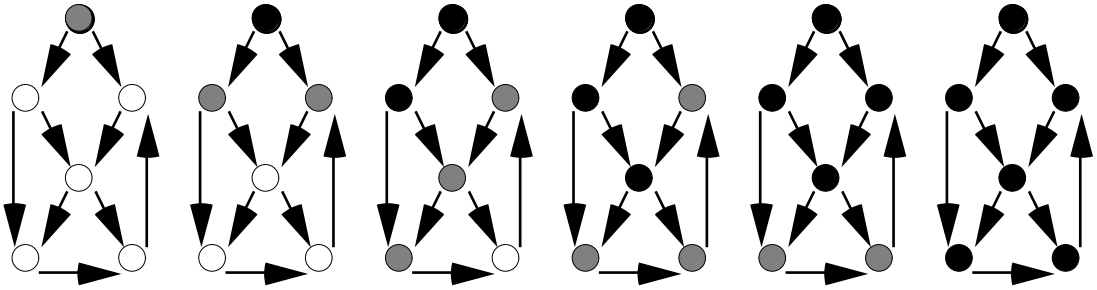
Exemple

J'ai choisi comme exemple le GC de SML (standard ML of New Jersey) dont le code, légèrement simplifié, apparaît dans l'encadré 5 (le code complet est expliqué en détail dans [App 89].)

On remarque qu'il ne se préoccupe que des objets accessibles, contrairement au GC à balayage, et qu'il n'est absolument pas récursif. Remarquons que le parcours du graphe se fait en largeur, ce qui est plutôt

encadré 4 : principe du coloriage à trois couleurs

Avec le coloriage à trois couleurs, on ne parcourt le graphe ni en largeur, ni en profondeur. Les objets qui seraient empilés (ou dont les pointeurs seraient inversés) dans un parcours en profondeur prennent ici la couleur grise : ce sont les objets pour lesquels il reste à colorier leurs *descendants* (les objets vers lesquels ils pointent.) On commence le parcours en grisant les racines, puis on prend un objets gris quelconque, et on grise les objets vers lesquels il pointe, puis on le noircit. On recommence jusqu'à la disparition complète des objets gris. On peut voir un exemple d'un tel parcours dans le dessin ci-dessous.
L'invariant de cet algorithme est qu'un objet noir ne pointe jamais directement sur un objet blanc.



1. On grise la racine

2. On noircit la racine et on grise ses fils.

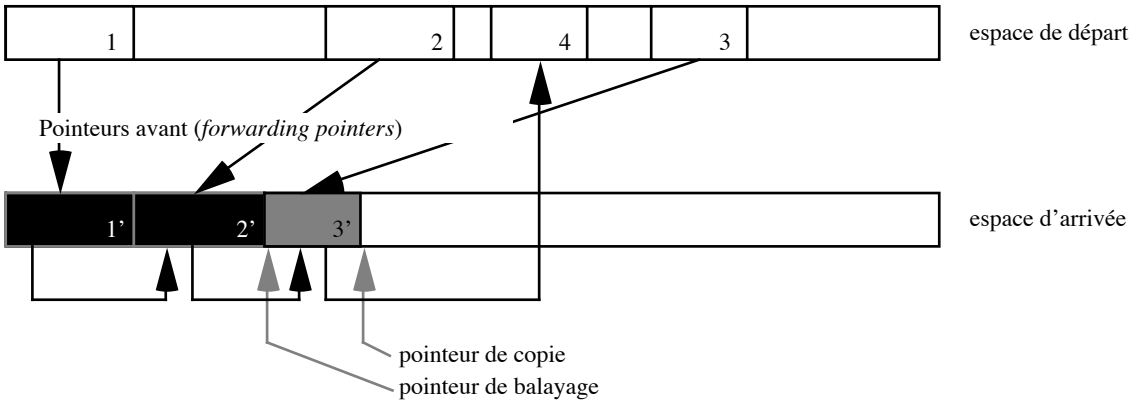
3. On noircit un des objets gris en grisant ses fils.

4. Si l'un des fils est déjà gris, il reste gris.

5. Si l'un des fils est noir, il reste noir.

6. A la fin, tous les objets accessibles sont noirs.

La mémoire au cours du GC :



Après copie de l'objet 4, on obtient :

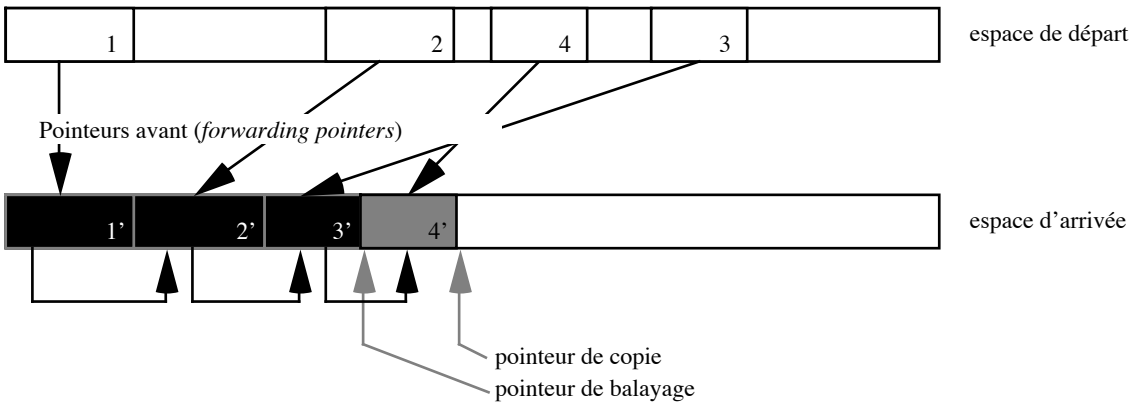


figure 4 : principe du GC à copie

encadré 5 : le GC de SML-NJ (en C)

```
gc(int *from_low, int *from_high,
   int *to_low, int *to_high,
   int **roots,
   int *(*get_more)()
  )
{ gmore=get_more; to_ptr = to_low;
  lowest=from_low; highest=from_high;

  while (*roots) forward(*roots++);

  { int *x = to_low;
  while (x<to_ptr)
  {int *p = x+1, descr = (*x), len=descr>>4;
    if (string_or_bytearray(descr))
          x += (len+3)/4 + 1;
    else {x += len+1;
        do {forward(p++);} while (p<x);
  }      }
}

forward(int *refloc)
{register int *m = *((int**)(refloc)), len;
 if( (m1)==0 && (m >= (int*)lowest && m < (int*)highest))
 { m--;  /* make m point at the descriptor */
   while (1)
     {len = (*m)>>4;
      switch(m&15)
       {case tag_backptr:
                m -= len;
                continue;
        case tag_string:
        case tag_bytearray:
            len = (len+3)/4;
            /* fall through */
        case tag_record:
        case tag_array:
            {int **i=(int**)m, **j=to_ptr;
                 while (j+len >= to_lim)
                      to_lim=gmore();
                 while (len-- >= 0)
                      {*j++ = *i++;}
                 ((int**)m)[1]= 1+(int*)to_ptr;
                 to_ptr = j;
            }
            (*m) = tag_forwarded;
            /* fall through */
        case tag_forwarded:
            *(int*)(reflog) += ((int*)m)[1] - ((int) (m+1));
            return;
        case tag_embedded:
        default:
                m--; continue;
}}   } }
```

mauvais pour la localité des références, dans un contexte de mémoire virtuelle. Cependant, cet effet est difficilement mesurable sous UNIX, et on choisit donc de ne pas en tenir compte.

Variantes

Les GC à copie ont donné lieu à beaucoup d'études théoriques, notamment pour des systèmes temps-réel, et en conjonction avec un système de générations. Il s'agit principalement de paralléliser l'algorithme à copie. Le détail des techniques utilisées se trouve en section 2.3.

2.2. Générations

Je vais d'abord exposer le principe dans sa variante la plus simple, pour énoncer ensuite les généralisations qu'on peut lui apporter.

On découpe la mémoire en deux zones appelées *générations*. Les objets de l'une sont appelés les *vieux*, et ceux de l'autre sont les *jeunes*. On alloue tous les nouveaux objets dans la zone des jeunes.

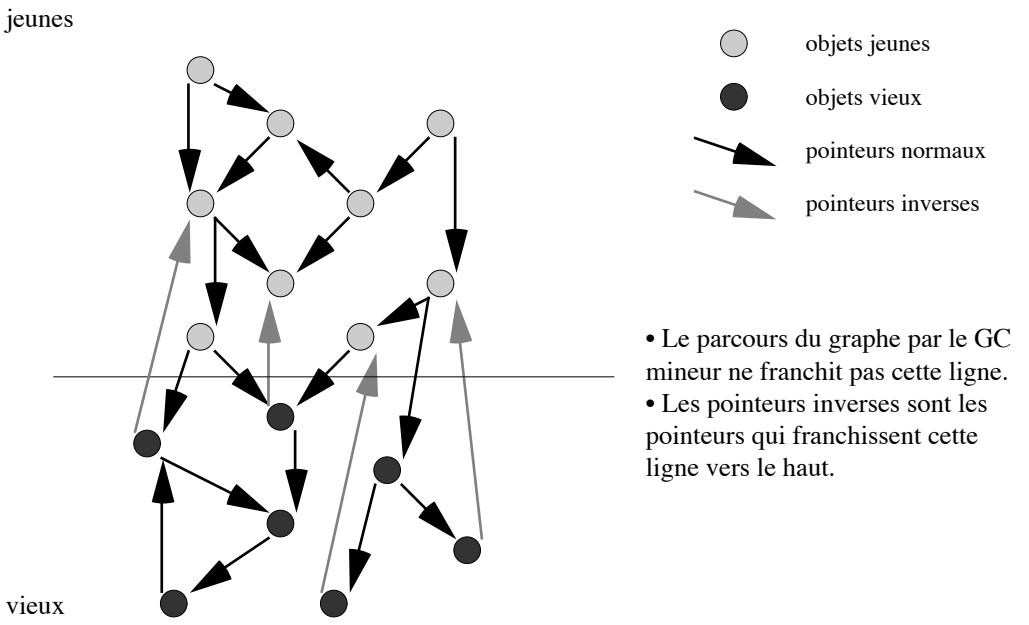


figure 5 : principe des générations (avec 2 générations)

Quand elle est pleine, on déclenche un *GC mineur*, qui copie dans la zone des vieux tous les objets jeunes accessibles.

Quand la zone des vieux est pleine, on déclenche un *GC majeur*, qui est simplement un GC normal, puisqu'il parcourt tout le graphe des objets accessibles.

L'intérêt du GC mineur c'est qu'il ne parcourt pas tout le graphe. En effet, les objets jeunes qui sont accessibles le sont soit à partir des racines, soit à partir des objets vieux qui pointent sur les jeunes. Or on prend soin de garder la trace des pointeurs des vieux vers les jeunes. Ce sont les *pointeurs inverses*. Le GC mineur utilise donc comme ensemble de racines les racines du mutateur, auxquelles on ajoute les pointeurs inverses (voir figure 5.)

Jusqu'à maintenant, le seul avantage des générations est qu'on peut faire un GC partiel sur les jeunes (c'est le GC mineur.) Mais il se trouve que les jeunes sont aussi ceux qui ont le plus fort taux de mortalité. En effet, l'expérience montre que la proportion d'objets qui survivent au GC mineur est très faible (de l'ordre de 1 à 3%[4].) Le GC mineur a donc très peu de choses à faire (recopier 3% de la zone jeune) pour récupérer un espace mémoire conséquent (la totalité de la zone jeune.) Il désalloue donc plus de 95% des objets, sans prendre trop de temps.

Un avantage supplémentaire du GC mineur : il ne dure pas très longtemps (1/100 à 1/10 de seconde, suivant les systèmes) et l'interruption est imperceptible dans un système interactif (les systèmes temps-réel ont des exigences plus strictes.)

Les généralisations possibles de ce mécanisme permettent d'augmenter le gain au prix d'une plus grande complexité du code : il s'agit principalement d'utiliser plus de deux générations. Un GC doit alors se faire sur toutes les générations plus jeunes qu'une génération donnée : c'est un GC plus ou moins majeur selon que la génération est plus ou moins vieille (voir figure 6.) Une version parallèle de cette méthode est étudiée dans [L&H 83]. On peut aussi exiger qu'un objet survive à plusieurs GC avant de le faire vieillir. Cette méthode permet de diminuer encore la proportion des objets qui migrent dans la vieille génération. Elle est exposée dans [Ung 84].

Le système des générations permet une amélioration miraculeuse des performances des GC, mais il demande quand même que quelques hypothèses sur le comportement du mutateur soient vérifiées :

• Les objets jeunes doivent avoir un fort taux de mortalité. C'est déjà vrai pour LISP et Smalltalk, dans lesquels la plupart des programmes utilisent beaucoup d'objets temporaires, et c'est encore plus vrai pour ML, où le système lui-même alloue une quantité incroyable d'objets pour les jeter presque aussitôt. Le mécanisme de générations est donc particulièrement adapté à ML. De plus, l'utilisation de structures intermédiaires est encouragée par un GC à génération, puisqu'ils sont peu coûteux ("Cheaper short term memory may improve programming style." [L&H 83])

• Le mutateur doit effectuer peu de modifications des objets déjà alloués, puisque c'est la seule source de pointeurs inverses. Lorsque le mutateur effectue

4. Ces chiffres sont donnés à titre d'exemple. Ce sont ceux que j'ai obtenus sur des exemples simples en ML.

une telle modification (qu'on appelle aussi *modification physique* ou *effet de bord*), il doit déterminer s'il y a création d'un pointeur inverse, et dans ce cas, le placer dans une table qui sera utilisée par le GC. Ce processus est assez coûteux. C'est le principal obstacle à l'utilisation des générations pour un langage paresseux, puisque les systèmes paresseux passent le plus clair de leur temps à faire de telles modifications. Dans [Ung 84], ce problème est résolu par le matériel (un bit de *tag* supplémentaire signale les pointeurs inverses.) Dans le cas de ML, cette hypothèse est parfaitement vérifiée, car de telles modifications sont découragées par le langage : les objets susceptibles d'être modifiés physiquement doivent être explicitement déclarés mutables, et ils sont soumis à des règles de typage plus strictes.

2.3. Algorithmes incrémentaux et parallèles

On peut envisager d'implémenter les algorithmes de GC en parallèle (le collecteur tourne en tant que processus distinct du mutateur, éventuellement sur un autre processeur) ou incrémentalement, c'est-à dire par petites tranches, en interrompant de mutateur souvent, mais pour de petites durées.

Le principal intérêt d'un GC parallèle est la suppression du temps de latence : le mutateur ne sera plus interrompu de façon imprévisible et incontrôlée. De plus, pour un système interactif, on peut profiter des temps morts (quand le système attend une entrée au clavier, par exemple) pour faire avancer le GC. Cette technique permet d'améliorer le confort d'utilisation pour un système interactif, et de mieux utiliser la puissance de calcul de la machine.

Cependant, il y a quelques problèmes à résoudre. Tout d'abord, il est nécéssaire de synchroniser le mutateur et le collecteur, puisqu'ils travaillent sur les mêmes données. Dijkstra montre aussi, dans [Dij 78] que le mutateur doit nécéssairement coopérer (c'est-à dire faire une partie du travail.) Les problèmes de synchronisation se posent de façon moins aiguë dans le cas incrémental, puisqu'on contrôle l'entrelacement des deux tâches.

Cependant, si on veut réaliser un système temps-réel, il faut étudier soigneusement la répartition du temps de calcul entre les deux processus, ce qui est plus facile avec un système parallèle.

En pratique, on peut appliquer cette transformation aux trois algorithmes de base, mais avec des degrés de difficulté différents.

• Pour le compte de références, il est naturellement incrémental, et on peut le paralléliser assez simplement. DeTreville a implémenté un tel système, dans lequel le mutateur signale au collecteur toutes les manipulations de pointeurs qu'il effectue, et le processus collecteur met à jour les compte et désalloue les objets en parallèle [DeT 88].

• Pour le balayage, il suffit de prendre l'ensemble des racines à un instant donné et de parcourir le graphe en partant de ces racines, pendant que le mutateur continue de le modifier. La synchronisation

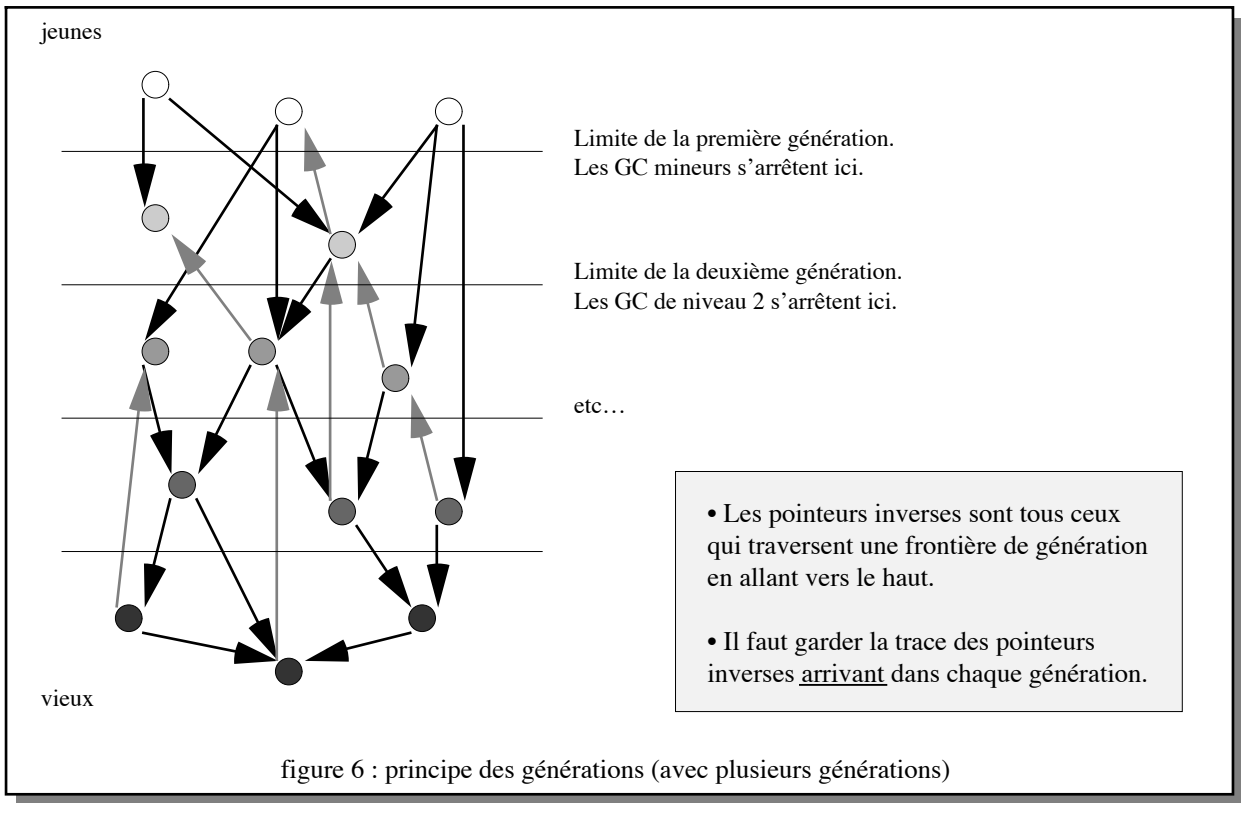


figure 6 : principe des générations (avec plusieurs générations)

nécessaire est peu importante, donc peu coûteuse.

• Dans le cas du GC à copie, c'est beaucoup plus difficile : quand on copie un objet, il faut mettre à jour tous les pointeurs vers cet objets, et le mutateur doit être bloqué pendant ce temps, qui est trop long (il faudrait parcourir tout le graphe pour trouver ces pointeurs.) Pour contourner ce problème, il existe plusieurs techniques, chacune avec ses défauts. On peut utiliser du matériel spécialisé, qui gère les pointeurs d'indirections; cette méthode est évidemment non portable. On peut utiliser le processeur de gestion de la mémoire (qui est un matériel assez répandu) pour déclecher une exception lors d'un accès à un objet pas encore copié; mais l'accès à ce circuit est encore coûteux. C'est la méthode préconisée par Appel, Ellis, et Li [ELA 88]. On peut utiliser un pointeur d'indirection systématique; ce qui est coûteux en mémoire comme en temps. C'est la technique utilisée par North et Reppy [N&R 87]. Enfin, on peut faire un test systématique lors de l'accès aux champs des objets (pour accéder à un objet, on commence par vérifier s'il a été copié) ce qui est très coûteux en temps (il s'agit de l'algorithme de Baker, et ses variantes, qui sont étudiées dans [Bak 78; L&H 83; [Daw 82; Hal 85].)

Toujours est-il que le coût supplémentaire d'un GC incrémental est assez élevé, et particulièrement dans le cas d'un GC à copie (même dans le cas d'un GC à balayage, l'incrémentalisation peut doubler le temps de calcul utilisé par le collecteur.)

2.4. Algorithme de Lang et Dupont

Cet algorithme est décrit en détails dans [L&D 88]. Son principe repose sur la remarque suivante : le GC à balayage et le GC à copie sont miscibles en toutes proportions. En effet, Bernard Lang et Francis Dupont ont proposé un algorithme abstrait qui généralise les deux méthodes, et qui peut ensuite être implémenté sous forme de GC à copie ou bien à balayage, mais aussi sous forme d'un GC qui copie certains objets et balaye les autres, la proportion pouvant varier d'un cycle à l'autre.

Voici le détail de l'algorithme abstrait :

On distingue trois ensemble d'objets dans la mémoire : les objets *parcourus*, les objets *marqués*, et les objets *non-marqués*, et on va déplacer les objets d'un ensemble dans l'autre pour trier les objets accessibles et inaccessibles (voir figure 7.)

Au début du GC, tous les objets de la mémoire sont dans l'ensemble des non-marqués. On fait passer les racines dans l'ensemble des marqués, puis on itère l'opération suivante jusqu'à ce qu'il n'y ait plus d'objet marqué : prendre un objet marqué, marquer tous ses fils non-marqués, et le mettre dans l'ensemble des parcourus. Quand on a fini, tous les objets accessibles (et eux seuls) sont dans l'ensemble des parcourus.

Remarquons qu'au cours de l'exécution de l'algorithme, on n'a jamais un objet parcouru qui pointe sur un objet non-marqué. Cet invariant est à la base de beaucoup de preuves d'algorithmes sur les GC.

Le GC à copie peut être considéré comme un cas particulier de cet algorithme : l'ensemble des objets non-marqués est l'espace de départ; l'ensemble des objets parcourus est la portion de l'espace d'arrivée qui se trouve avant le pointeur de balayage (en noir sur la figure 4), et l'ensemble des objets marqués est la portion de l'espace d'arrivée comprise entre le pointeur de copie et le pointeur de balayage (en gris sur la figure 4.)

Dans un GC à balayage, l'ensemble des objets parcourus est représenté par les objets noirs, l'ensemble des objets non-marqués par les objets blancs, et l'ensemble des objets marqués est dans la pile (parcours récursif), dans la branche des pointeurs inversés (inversion de pointeurs), ou bien dans les objets gris (coloriage à trois couleurs.)

Enfin, on peut couper la mémoire en trois sous-espaces : l'espace de départ, l'espace d'arrivée et l'espace de balayage. Quand on doit marquer un objet, on le copie dans l'espace d'arrivée s'il est dans l'espace de départ, et on le colorie en gris s'il est dans l'espace de balayage : c'est l'algorithme de GC de Lang et Dupont, qui mélange donc copie et balayage.

Ses avantages sont ceux des GC à copie **et** ceux des GC à balayage : il compacte la mémoire, il peut aller aussi vite que le GC à copie, et cependant il peut laisser plus de la moitié de la mémoire à la disposition du mutateur. Il compacte la mémoire incrémentalement, au fur et à mesure que l'espace de départ se déplace (voir encadré 8, page 18.)

Ses inconvénients sont aussi ceux des GC à copie **et** ceux des GC à balayage : il n'est pas facile à paralléliser (puisqu'il déplace certains objets) et l'allocation exige une liste libre (puisqu'il ne compacte pas toujours toute la mémoire.) De plus, sa mise en œuvre est plutôt compliquée, comme on le verra.

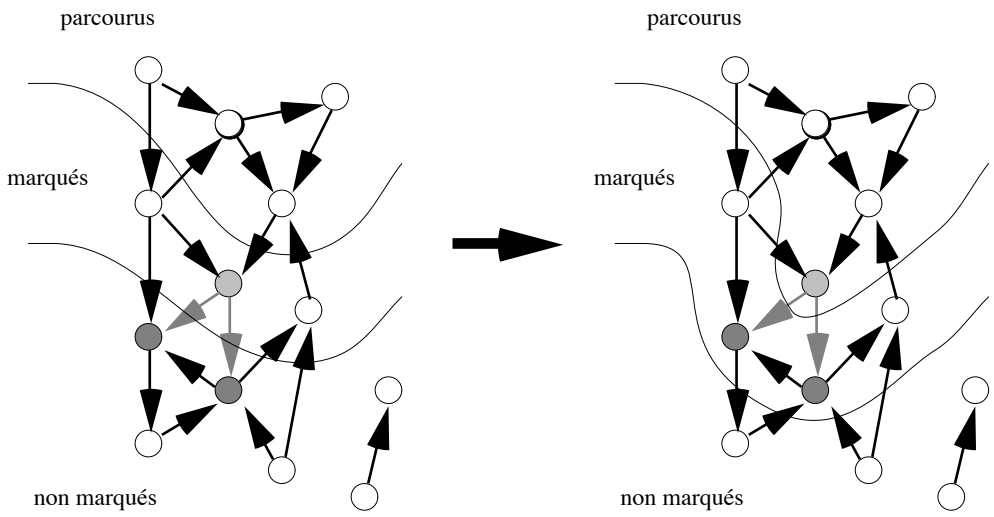


figure 7 : parcourir un objet en marquant ses fils

2.5. Comparaison des différents algorithmes.

Voici (dans l'encadré 6) un tableau récapitulatif des quatres principaux algorithmes de GC.

Les critères retenus sont :

• Le coût en temps de calcul.

• Le coût en mémoire.

• La méthode d'allocation : linéaire ou en liste libre; la première méthode étant la plus rapide, surtout si on a des objets de tailles variables, ce qui est le cas en ML.

• La possibilité d'adapter un système de générations (qui est un avantage important.)

• La possibilité de paralléliser ou de rendre incrémental le GC; elle est capitale pour les systèmes temps-réel, et intéressante pour les applications interactives.

• La facilité d'implémentation.

encadré 6 : tableau récapitulatif

algorithme	coût en temps	coût en mémoire	allocation	générations	parallèle / incrémental	mise en œuvre
compte de références	élevé	faible	liste	oui (3)	facile	*** (6)
balayage	assez élevé	moyen	liste	oui (3)	assez facile	**
copie	faible	élevé	linéaire	oui	difficile	*
Lang & Dupont	variable (1)	variable (1)	liste (2)	oui (4)	possible (5)	****

Remarques :

(1) En faisant varier les proportions, on peut faire varier ces caractéristiques entre celles d'un GC à copie et celles d'un GC à balayage.

(2) Il est possible d'obtenir des performances quasiment identiques à celles d'une allocation linéaire.

(3) On peut utiliser un algorithme quelconque sur la génération vieille (la plus volumineuse) mais il faut un GC à copie pour les autres.

(4) A ma connaissance, ceci est la première implémentation d'un GC de Lang et Dupont avec des générations.

(5) On peut paralléliser assez facilement avec une proportion faible d'objets copiés (voir 1.)

(6) Compte tenu de la nécéssité d'un GC auxiliaire.

3. Implémentation

3.1. Choix stratégiques

J'ai choisi d'implémenter l'algorithme de Lang et Dupont avec un mécanisme de générations, pour les raisons suivantes :

- C'est un algorithme récent et qui n'avait jamais été utilisé avec des générations; il s'agit donc d'une innovation intéressante.
- L'algorithme est assez prometteur pour ce qui est de la vitesse, et il se comporte bien dans le cas où la mémoire est presque pleine, c'est-à dire quand les objets actifs occupent plus de 30% du tas. (Dans ces conditions, l'algorithme à copie est trop coûteux.)

Pour implémenter le GC, j'ai choisi le langage C [K&R 88]. Ce choix satisfait les critères suivants :

- La portabilité. Le langage C est porté sur un grand nombre de machines.
- L'efficacité. C est un langage d'assez bas niveau pour obtenir des programmes efficaces après compilation.

3.2. Le modèle mémoire

Je vais maintenant détailler le modèle mémoire choisi, c'est-à dire l'ensemble des formats et des codages utilisés pour représenter les données.

Architecture des machines modernes

Le programme fait certaines hypothèses sur la machine. (Ces hypothèses sont assez faibles pour être vérifiées sur la plupart des machines modernes : 680x0, 32032, 80x86, RISC.)

- La mémoire est organisée en octets.
- Les adresses sont stockées sur 32 bits (4 octets.)
- Un *mot* est constitué de 4 octets consécutifs commençant à une adresse multiple de 4.

Le programme garantit aussi que tous les objets sont constitués d'un nombre entier de mots. Dans ces conditions, seuls les 30 bits de poids fort d'un pointeur sont significatifs, puisque tous les pointeurs sont multiples de 4. Cette propriété est importante pour la suite.

Valeurs

Comme le fait remarquer Appel dans [App 89], la compilation des fonctions polymorphes est grandement facilitée si tous les objets manipulés par le programme ont la même taille. On fait en ML comme en LISP, c'est-à dire qu'on décide que tous les objets ont la taille d'un pointeur. Si un objet est trop grand pour être représenté de cette façon, on l'alloue dans le tas et on utilise un pointeur sur lui.

Il y a donc deux classes de *valeurs* : les valeurs *immédiates* et les pointeurs. Comme le GC doit éviter de modifier les objets immédiats, il faut lui donner le moyen de distinguer ces deux classes. On utilise un mécanisme similaire à celui décrit dans [App 89] pour SML et dans [G&R 83] pour Smalltalk : un ou deux bits de tag permettent de faire la différence; de plus, comme tous les pointeurs sont multiples de 4, on peut le faire sans réduire l'espace d'adressage. Il y a deux schémas possibles (voir figure 8) et le code du GC est prévu pour s'adapter aux deux : il suffit de changer une macro dans le programme.

Blocs

Un bloc est composé d'un *en-tête*, et de un ou plusieurs champs (voir encadré 7.) Le GC utilise l'information de taille et le tag contenues dans l'en-tête (ainsi que la couleur.) Si le tag est inférieur ou égal à `NO_SCAN_TAG`, le bloc est un bloc *structuré*. Dans un bloc structuré, le contenu est composé de champs (chaque champ occupe un mot) qui doivent tous être des valeurs (immédiats ou pointeurs.) Dans un bloc non structuré, le mutateur peut placer une donnée quelconque (un nombre entier de mots) à condition qu'elle ne contienne pas de pointeur (le contenu des blocs non structurés est ignoré par le GC.)

Les blocs structurés sont par exemple les cellules de liste, les vecteurs, les fermetures; les blocs non structurés servent aux chaînes de caractères.

Zones

La mémoire est découpée en trois zones : la *zone statique*, la *zone majeure* (ou *tas majeur*), et la *zone mineure* (ou *tas mineur*). La zone mineure et la zone majeure ensemble constituent le *tas* (voir figure 9.)

Le tas est entièrement géré par le gestionnaire de mémoire. Il ne contient que des blocs, l'un après l'autre, sans espace entre eux.

La zone statique est gérée par allocation et désallocation explicite, ce qui signifie que le GC n'a pas à l'explorer. Elle est destinée à contenir les objets à très longue durée de vie, ainsi que les objets qu'il est difficile de déplacer, par exemple le code.

On peut trouver dans le tas des pointeurs vers la zone statique (ils seront traités comme des constantes par le GC, puisqu'il ne déplace que des objets du tas) mais la réciproque n'est pas vraie, car les objets du tas peuvent bouger, et le GC doit contrôler tous les pointeurs vers les objets qu'il peut déplacer.

Accès aux structures de données

Le GC appelle la macro `IS_BLOCK` pour savoir si un mot particulier est un pointeur ou non.

Il utilise les macros `TAG_HEADER`, `SIZE_HEADER`, et `COLOUR_HEADER` pour accéder aux champs de l'en-

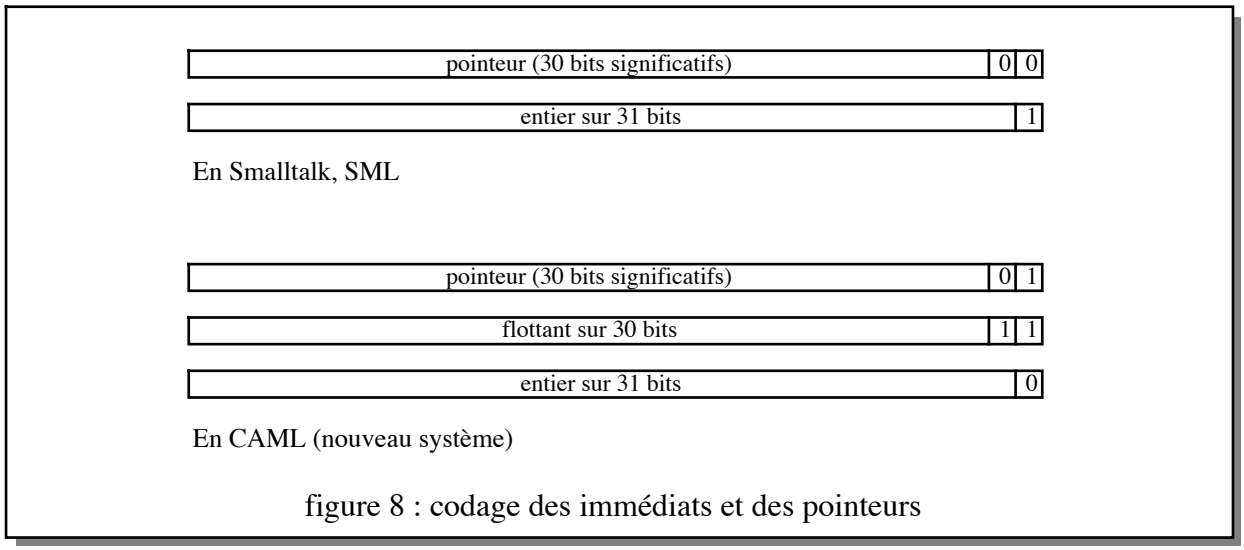


figure 8 : codage des immédiats et des pointeurs

tête.

Comme la représentation des pointeurs et des immédiats peut varier, on utilise la macro `IS_BLOCK` pour savoir si une valeur est un pointeur, et `MLINT`, `CINT`, `ADDRESS`, et `OBJECT` pour convertir les valeurs (codées suivant la description ci-dessus) en données directement exploitables par le programme C.

Toutes ces macros sont définies dans les fichiers “header.h” et “values.h” qui regroupent les définitions dépendant du format choisi pour le codage des valeurs, des blocs, et des en-têtes.

Services offerts

Les services offerts par le gestionnaire de mémoire sont les suivants :

• Allocation d’un objet ordinaire (dans la zone mineure.)

• Allocation d’un “gros” objet (trop gros pour la zone mineure) avec ou sans initialisation.

• Allocation en zone statique d’un morceau de mémoire[5] (pas forcément un objet.)

• Recopie en zone statique d’une structure de données[5].

• Modification à la demande de la taille de la zone mineure[5].

Obligations du mutateur

Le mutateur doit obéir aux trois règles suivantes :

• Il doit utiliser la macro fournie par le gestionnaire de mémoire pour effectuer toute modification d’un objet structuré alloué dans le tas.

• Il ne doit pas garder de pointeur infixe lors d’un appel au GC. (Un pointeur *infixe* est un pointeur qui pointe à l’intérieur d’un objet au lieu de pointer sur l’en-tête. Dans ce cas, le GC n’a aucun moyen de retrouver l’en-tête et d’identifier l’objet pointé.)

• Enfin, comme pour tout algorithme de GC, il ne doit garder aucun pointeur vers le tas qui ne soit connu du collecteur. (Cette propriété rend parfois difficile l’optimisation de code, puisque les valeurs ne peuvent pas être placées dans n’importe quel registre.)

Le gestionnaire de mémoire est capable d’agrandir la zone statique pour satisfaire toutes les requêtes : il suffit de déclencher un GC majeur avec comme espace de départ la partie du tas majeur qui est contiguë à la zone mineure pour libérer cette partie de la mémoire[5].

3.3. Implémentation de l’algorithme de Lang et Dupont

Une fois muni d’un bon algorithme (Lang et Dupont, générations) et d’un modèle mémoire simple mais robuste, il ne restait plus qu’à coder l’algorithme.

Avant de détailler le fonctionnement du programme, voici quelques problèmes rencontrés et les solutions retenues.

Il existe deux méthodes pour tracer à partir d’un ensemble de racines : marquer toutes les racines puis tracer tout le graphe; ou bien tracer un sous-graphe aussi grand que possible à partir de chaque racine.

La première méthode a l’avantage d’être plus simple et plus rapide (on évite un appel de fonction pour chaque racine) tandis que la seconde donne, pour un GC qui copie les objets, de meilleurs résultats quant à la localité des références (voir figure 10.) J’ai choisi la deuxième solution, puisqu’il me semble que le nombre de racines est assez faible par rapport au nombre d’objets. Ce problème n’a cependant pas fait l’objet d’études précises.

5. Ce trait n’est pas encore implémenté, mais il ne pose pas de problème particulier.

encadré 7 : la structure d'un bloc

Un bloc est constitué d'un nombre entier de mots.

Il commence par un en-tête qui contient :
• La taille du bloc (en mots.)
• Le tag (qui peut être réduit à 1 bit.)
• La couleur (qui n'est utilisée que par le GC.)
• Des drapeaux (facultatifs, ils ne sont pas utilisés par le GC.)

Ensuite, on a les champs, qui occupent un mot chacun (si le tag indique un bloc structuré) ou bien un nombre entier de mots contenant des données quelconques :

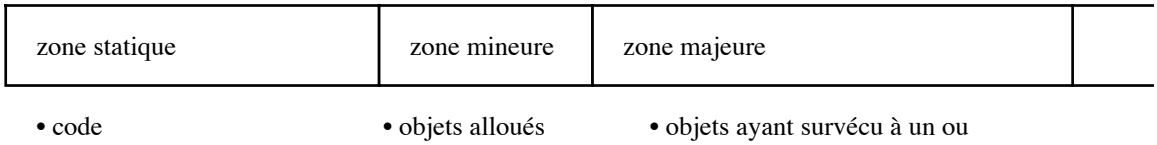
bloc structuré bloc non structuré

Le choix de la taille et de l'emplacement de l'espace de départ et de l'espace d'arrivée doivent être faits soigneusement. J'utilise une heuristique qui satisfait les conditions suivantes :

- l'espace de départ est en un seul bloc, et l'espace d'arrivée aussi, ce qui permet d'avoir des tests d'appartenance peu coûteux.
- Lorsque le taux d'occupation du tas est faible, l'espace de balayage est aussi réduit que possible (la fragmentation peut empêcher de le réduire à zéro en un seul cycle.)
- Lorsque l'espace de balayage est réduit à zéro, l'espace de départ et d'arrivée sont échangés à chaque cycle, comme dans un GC à copie, ce qui fait que le GC reste en mode "copie pure" tant que le taux d'occupation du tas le permet.

L'algorithme de Lang et Dupont est théoriquement aussi efficace que l'algorithme à copie, tant qu'il reste en mode "copie pure"; il faut cependant étudier soigneusement les tests d'appartenance pour que cette propriété reste vraie dans l'implémentation. En effet, il faut tester pour chaque objet s'il appartient à l'espace de départ avant de le copier. On a un arbre de possibilités pour chaque objet, et en échangant certains nœuds, on peut raccourcir la branche la plus probable (voir figure 11.) J'ai optimisé le programme pour favoriser le cas "copie pure" : il ne fait que deux tests contre quatre pour l'algorithme naïf (deux tests pour l'appartenance au tas majeur, et deux autres pour l'appartenance à l'espace de départ.)

L'espace de départ pose un problème épineux : comment repérer où il s'arrête. A priori, il a la même taille que l'espace d'arrivée, mais il est impératif qu'il contienne un nombre entier d'objets (en effet, on ne peut pas copier la moitié d'un objet et marquer l'autre moitié.) On ne peut pas balayer l'espace pour découvrir le premier objet qui déborde, ce serait trop coûteux en temps. La solution retenue coûte un test supplémentaire pour chaque objet de l'espace de balayage, mais favorise encore le cas "copie pure".



• code • constantes structurées • objets statiques • variables globales du système (dont celles du GC)	• objets alloués récemment (jeune génération)	• objets ayant survécu à un ou plusieurs GC (vieille génération) • objets trop gros pour être alloués dans la zone mineure

figure 9 : organisation de la mémoire

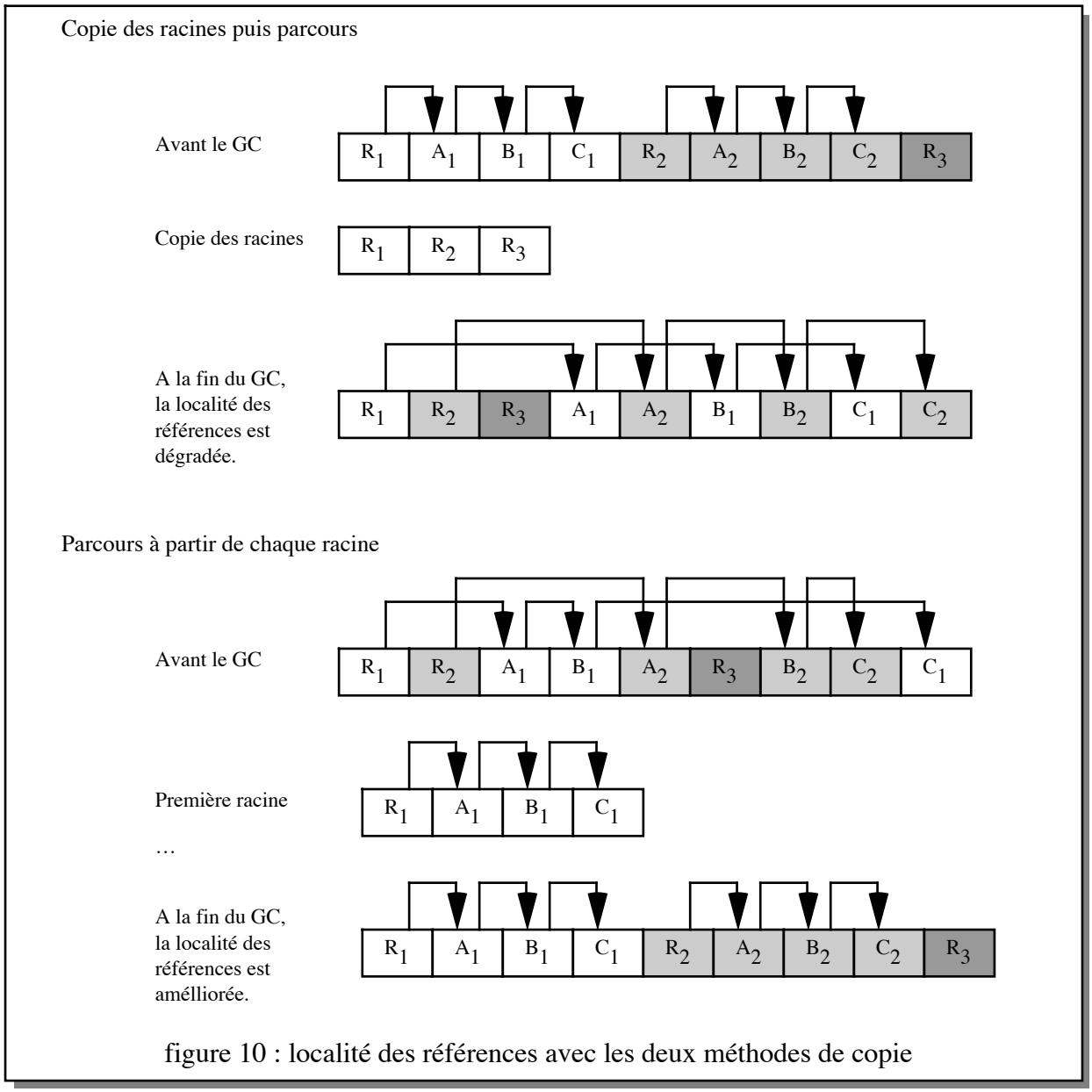


figure 10 : localité des références avec les deux méthodes de copie

Le parcours des objets dans l'espace de balayage s'effectue avec l'algorithme des trois couleurs. Celui-ci est cependant très inefficace en temps, puisqu'il exige de faire de nombreux balayages de la mémoire pour chercher les objets gris. J'ai donc implémenté un "cache" qui est une pile de longueur limitée. Le parcours se fait comme un parcours avec pile, et si la pile déborde, on colorie l'objet en gris pour pouvoir le retrouver plus tard. Cette amélioration est très rentable (de l'ordre d'un facteur 100) quand le GC fonctionne avec le maximum de balayage.

L'allocation doit se faire avec une liste libre, puisque l'espace libre n'est pas toujours contigu. Cependant, on obtient des performances comparables à celles d'une allocation linéaire en remarquant que (surtout dans le cas "copie pure") les blocs de mémoire libre sont beaucoup plus gros que les requêtes d'allocation, et en utilisant une méthode d'allocation linéaire dans chacun de ces blocs.

Enfin, le dernier problème qui se pose est celui de la taille et de la complexité du code du GC. Les deux sont nettement plus grandes que pour un GC à copie, pour des performances comparables. La taille du code n'est pas encore un problème (moins de 10ko sur une station SUN) par rapport à un système ML complet (plusieurs Mo en moyenne.) La complexité du code pose un problème de fiabilité, que seule l'utilisation peut trancher.

Voici une description détaillée du programme (le code est donné en annexe.)

La détermination des espaces de départ et d'arrivée est faite dans **memory.c**. Elle suit le schéma général

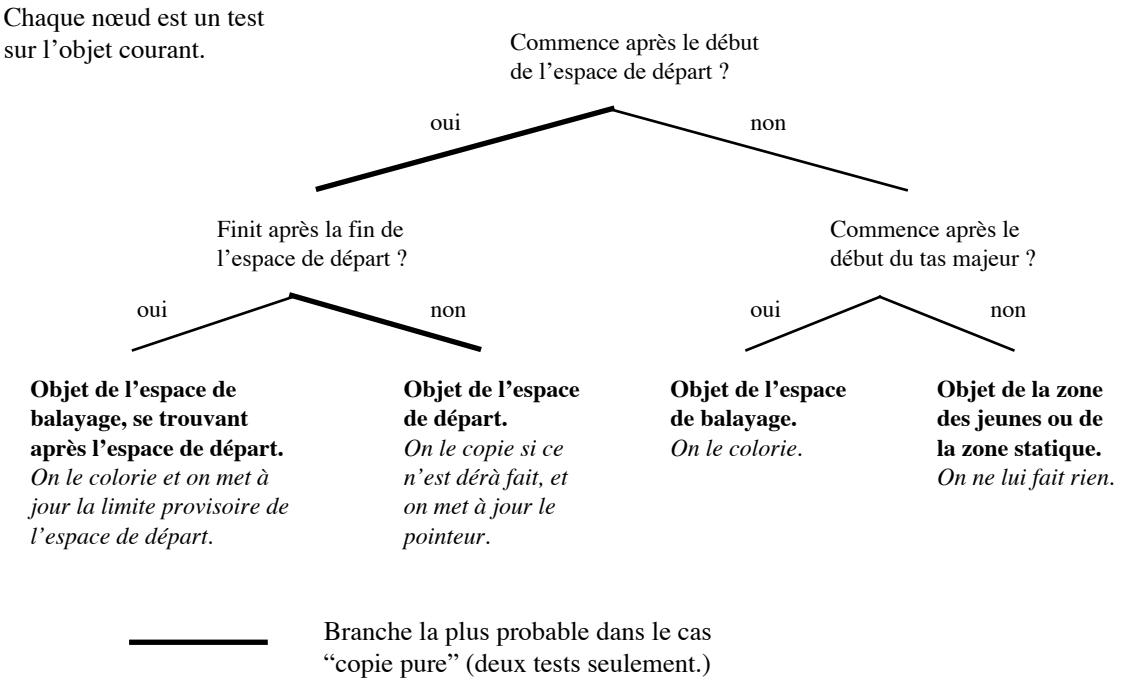


figure 11 : l'arbre de décision de la macro `MARK_OR_COPY`

exposé dans l'encadré 8, mais il faut tenir compte de tous les cas particuliers : si on se trouve à la fin du tas, s'il y a un objet plus gros que l'espace de départ … Remarquons aussi que l'espace d'arrivée est déterminé à la fin du cycle précédent, tandis que l'espace de départ est fixé juste avant le GC. En effet, l'espace d'arrivée est réservé, il ne faut pas allouer dedans, donc il faut le fixer avant de donner la liste libre au mutateur; tandis que l'espace de départ peut dépendre du contenu du tas, qui n'est connu qu'au moment du déclenchement du GC.

L'algorithme de Lang et Dupont se trouve dans le fichier **gc-major.c**; il comporte essentiellement deux phases : d'une part marquage et copie, et d'autre part balayage.

La première phase consiste à appeler la fonction `trace_from` pour chaque racine.

Cette fonction commence par marquer la racine (en appelant `MARK_OR_COPY`) puis elle entre dans une boucle de recherche puis parcours d'un objet marqué. La recherche se fait par la macro `FAST_FIND_GRAY`. Il s'agit d'une recherche "rapide", c'est-à dire qu'on cherche dans la FIFO de l'espace d'arrivée et dans le "cache" qui contient des objets gris, mais on évite de faire un balayage complet de l'espace de balayage pour chercher un objet gris; on fera ce balayage seulement quand toutes les racines auront été traitées.

`Trace_from` appelle alors `MARK_OR_COPY` pour chaque objet renvoyé par `FAST_FIND_GRAY`.

La macro `MARK_OR_COPY` constitue le cœur du GC. Elle détermine ce qu'il convient de faire d'un objet : le copier et mettre à jour la référence s'il est dans l'espace de départ, mettre à jour la référence s'il a déjà été copié, le colorier en gris s'il est dans l'espace de balayage. De plus, elle met à jour la variable `first_after_from_space` qui indique le premier objet accessible qui se trouve après l'espace de départ, pour permettre de fixer la frontière de celui-ci entre deux objets. Enfin, elle effectue divers ajustements des variables du GC mineur (qui seront détaillés plus loin) dans le cadre du mécanisme de générations.

Enfin, la macro `GRAYEN` prend en charge les objets accessibles de l'espace de balayage. Elle distingue trois cas : d'abord, les objets non structurés : on peut tout de suite les noircir; ensuite les objets structurés, s'il y a encore de la place dans le cache : on les met dans le cache et on les noircit tout de suite (le fait qu'ils soient dans le cache assure qu'on va colorier leurs fils); enfin, s'il n'y a plus de place dans le cache, on grise l'objet, et on incrémente la variable `cache_gray_overrun` qui compte le nombre d'objets gris dispersés dans le tas.

La première phase se finit avec une dernière boucle "chercher un objet marqué et le parcourir" qui cherche tous les objets gris laissés dans le tas.

La deuxième phase est très simple en comparaison : après avoir ajusté la taille de l'espace de départ, on itère sur tous les objets de l'espace de balayage (avec

`first_in_ms_space` et `FIND_NEXT_IN_MS_SPACE`) pour blanchir ceux qui sont noirs (en prévision du cycle suivant) et placer dans la liste libre ceux qui sont blancs (c'était le but de toute l'opération.) Il est à noter que la liste libre peut s'agrandir d'un autre bloc après la réservation du prochain espace d'arrivée : l'excédent d'espace libre dégagé par la copie de l'espace de départ (cet espace libre se trouve à la fin de l'espace d'arrivée.)

3.4. Le mécanisme de générations

Commençons par remarquer qu'il peut être intéressant de de **pas** avoir de GC, du moins pour certains objets. En effet, le code et les constantes qu'il utilise ont la même durée de vie que le système lui-même. Il est donc parfaitement inutile de perdre du temps à essayer de récupérer l'espace qu'ils occupent. C'est particulièrement vrai dans le cas d'un système interactif où on recopierait 1Mo de code et de structures de données du compilateur à chaque GC. (Le compilateur CAML est sans doute le plus gros programme CAML jamais écrit.) D'où l'idée d'une zone statique, c'est-à dire dont le GC ne s'occupe pas. délicat à faire, puisque le compteur de programme est un pointeur infixe dans le code (ce problème est résolu dans [App 89], au prix d'un test supplémentaire par objet copié), et plutôt inutile, puisqu'on désalloue très rarement le code.

On a donc trois zones : la zone statique, la zone majeure, la zone mineure (voir figure 9.) On fait des GC souvent sur la zone mineure, rarement sur la zone majeure, et jamais sur la zone statique. Le principe est simple : on alloue dans la zone mineure. Quand elle est pleine, le GC mineur se déclenche. Il copie les objets accessibles de la zone mineure dans la zone majeure. Si celle-ci est pleine, on déclenche un GC majeur et on continue ensuite le GC mineur qui a déclenché le GC majeur.

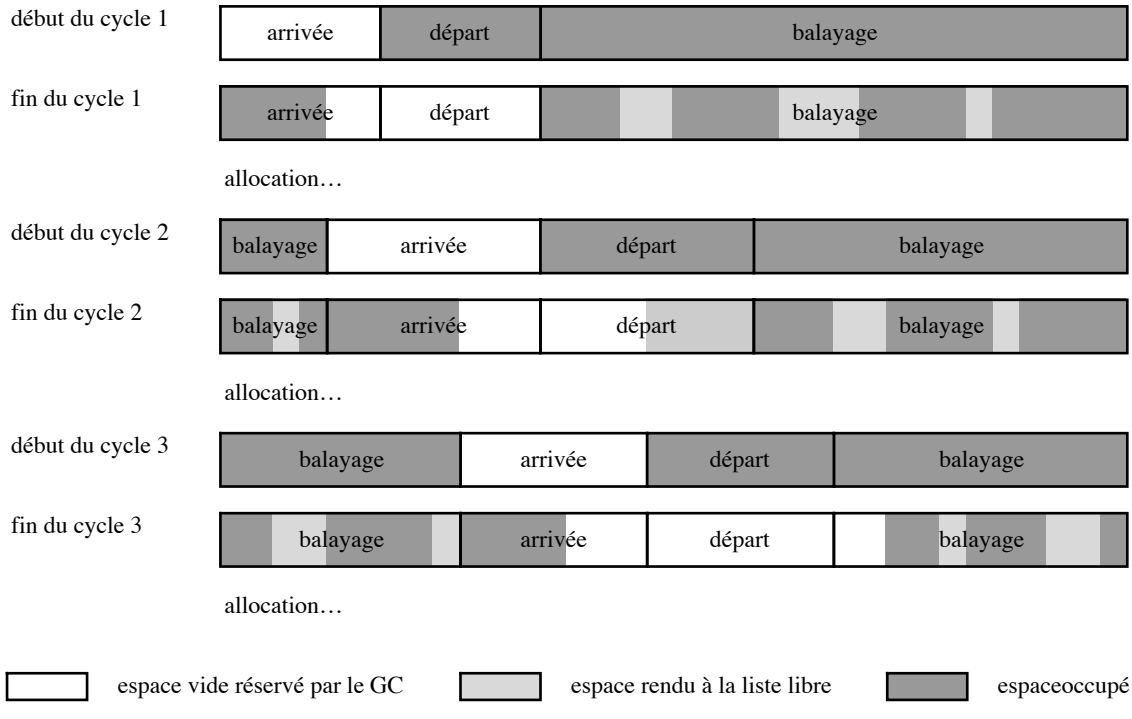
Le GC mineur est très simple (c'est un algorithme à copie) mais son interaction avec le GC majeur est une mécanique assez délicate.

Le GC mineur utilise deux tables de taille fixe pour travailler : la table des pointeurs inverses et la table des objets copiés.

La table des pointeurs inverses (`ref_table`) recense tous les pointeurs qui pointent dans le tas mineur depuis le tas majeur (voir figure 12.) Elle est mise à

encadré 8 : détermination des espaces de départ et d'arrivée

Les espaces de départ et d'arrivée effectuent une rotation au cours du temps. L'espace d'arrivée est constitué de l'espace de départ du cycle précédent (il a été complètement libéré par le GC précédent) et l'espace de départ se trouve juste après. Ainsi, on peut recoller la partie inutilisée de l'espace d'arrivée avec l'espace de départ, pour obtenir un nouvel espace d'arrivée plus important.



De plus, on évite de déplacer le code, ce qui est assez

jour par la macro `MODIFY` que le mutateur appelle pour toute modification d'un objet du tas. Elle est utilisée comme ensemble de racines supplémentaires par le GC mineur. Quand le GC majeur se déclenche, il doit reconstruire cette table au fur et à mesure qu'il fait son parcours, puisque les pointeurs de cette table ne sont pas des valeurs.

La table des objets copiés (`copied_young`) est la FIFO qui permet de parcourir les objets marqués. L'espace où on copie les objets ne peut pas jouer ce rôle, puisqu'il n'est pas en un seul morceau. On stocke donc dans cette table un pointeur sur chaque objet copié par le GC mineur, pour pouvoir copier ses fils ensuite. Cette table est simplement mise à jour par le GC majeur. (Elle est utilisée comme ensemble de racines spéciales.)

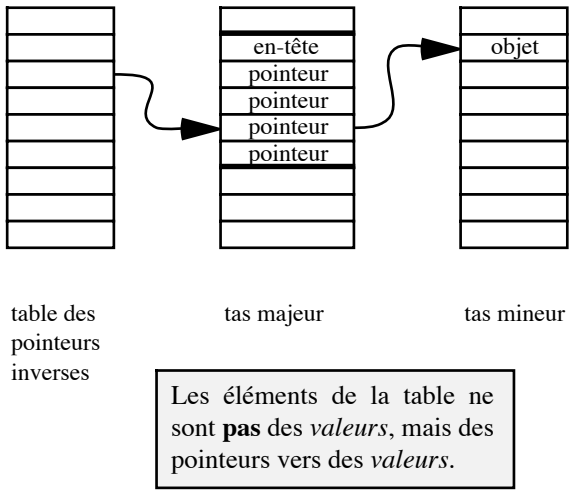


figure 12 : table des pointeurs inverses

Il faudrait essayer deux variantes intéressantes qui font intervenir une génération intermédiaire, sans toutefois compliquer excessivement le code (voir figure 13.) Cela permet d'espacer encore plus les GC majeurs puisqu'on réduit le nombre d'objets qui arrivent dans la zone majeure.

On pourrait aussi envisager une ségrégation entre les objets mutables et les non-mutables : les objets mutables seraient placés dans une zone spéciale, explorée à chaque GC mineur (puisque ces objets peuvent pointer vers les jeunes.) L'avantage de cette méthode, qui est décrite dans [Spi 89; H&M 86], est de réduire le surcoût qui affecte la modification d'un objet alloué, mais elle exige une zone de plus, avec

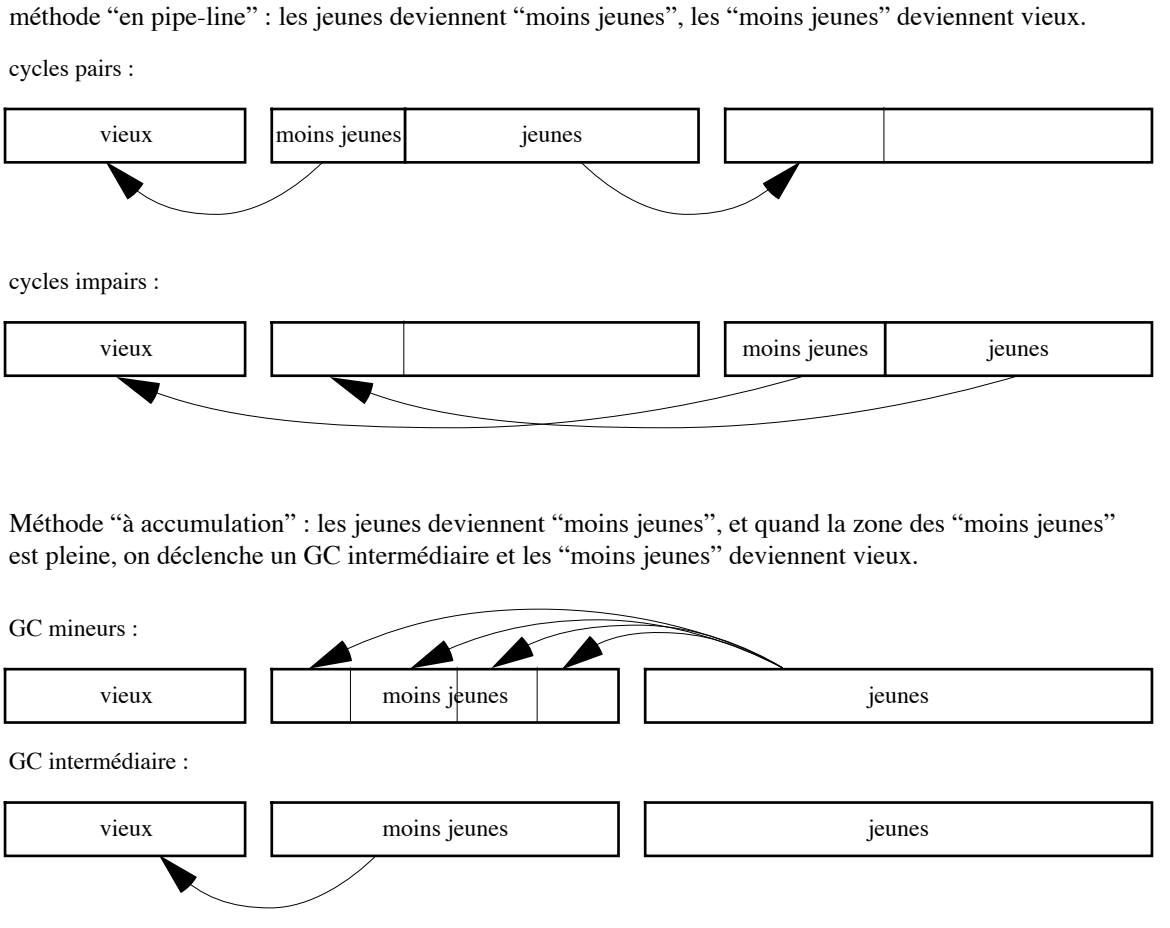


figure 13 : deux façons d'ajouter une génération intermédiaire

tous les problèmes sous-jacents (que faire quand elle déborde, comment l'étendre, comment la déplacer ?)

Il reste un problème ennuyeux : le GC mineur passe une bonne partie de son temps à parcourir la pile à la recherche de racines (environ 50% de son temps.) La solution adoptée pour SML est la suppression pure et simple de la pile : les blocs d'appel de fonction (qui contiennent les arguments et les variables locales de la fonction appelée) sont alloués dans le tas, sous prétexte que cette allocation est peu coûteuse [App 89]. Une autre solution consisterait à repérer la partie de la pile qui n'a pas changé depuis le dernier GC mineur (avec la complicité du circuit de gestion de la mémoire, par exemple.) Je n'ai pas retenu cette solution car elle n'est pas portable.

4. Performances

On s'intéresse ici aux performances du GC majeur, étant entendu que le mécanisme de générations apporte un gain important.

Je vais d'abord étudier brièvement le comportement des GC à copie et à balayage en fonction du taux d'occupation du tas (c'est-à dire en fonction du rapport entre le nombre de mots appartenant à des objets actifs et la taille du tas.) Puis je ferai la même étude pour le GC de Lang et Dupont, et enfin je comparerai les trois.

Toutes les courbes présentées ont été obtenues sur une station SUN 3/60, avec le même mutateur : un programme qui alloue alternativement dans deux listes des petits blocs de tailles aléatoires, et qui libère chaque liste en entier, alternativement, tout en maintenant constant le nombre d'objets actifs. L'utilisation de deux listes permet d'obtenir un taux de fragmentation non nul dans le tas. Les temps

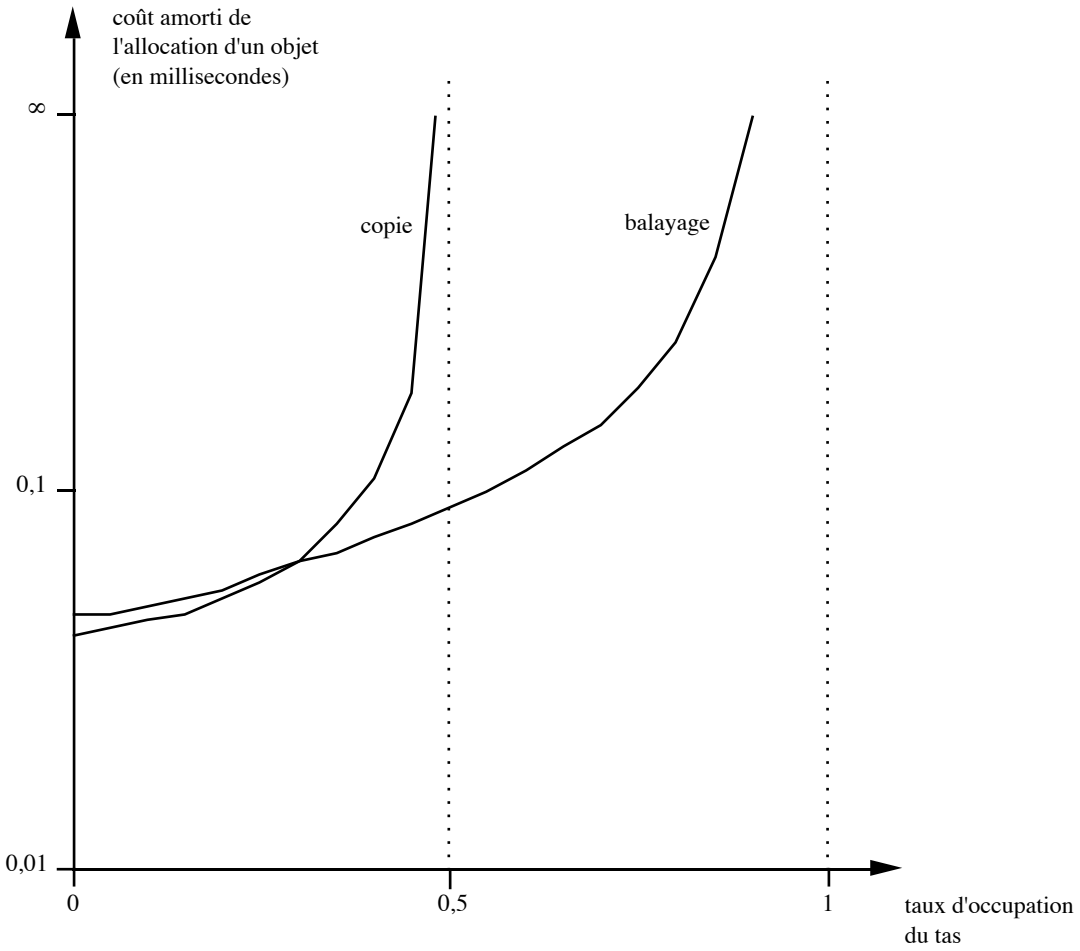


figure 14 : coût des GC à copie et à balayage

calculés sont des coûts amortis d'allocation, c'est-à dire le coût de l'allocation, plus la fraction du coût du GC qu'elle a contribué à déclencher. Il s'y ajoute aussi le coût des manipulations de listes, qui est difficile à mesurer séparément, et qui rend les mesures imprécises lorsque le coût du GC est faible.

Il est à remarquer que les chiffres obtenus sont indépendants de la taille du tas.

La figure 14 présente les courbes du coût en fonction du taux d'occupation pour le GC à copie et le GC à balayage.

Le GC à copie.

Pour l'algorithme à copie, le GC est peu coûteux lorsque le taux d'occupation est faible, puisque le coût d'un GC est proportionnel au nombre d'objets actifs. Le coût augmente avec le taux d'occupation, pour deux raisons : d'abord, chaque GC est plus coûteux, car il y a plus d'objets à copier, et ensuite, chaque GC récupère moins de place, donc les GC sont plus fréquents. On remarque que le coût amorti du GC à copie tend vers l'infini quand le taux d'occupation du tas tend vers 1/2.

Le GC à balayage.

En ce qui concerne le GC à balayage, la courbe a la même forme, mais plus aplatie. Elle démarre plus haut, car le coût du GC à balayage est proportionnel à la taille du tas plutôt qu'au nombre d'objets actifs, mais elle ne tend vers l'infini que pour un taux d'occupation atteignant 1. Elle passe donc en dessous de la courbe du GC à copie, c'est-à dire que le GC à

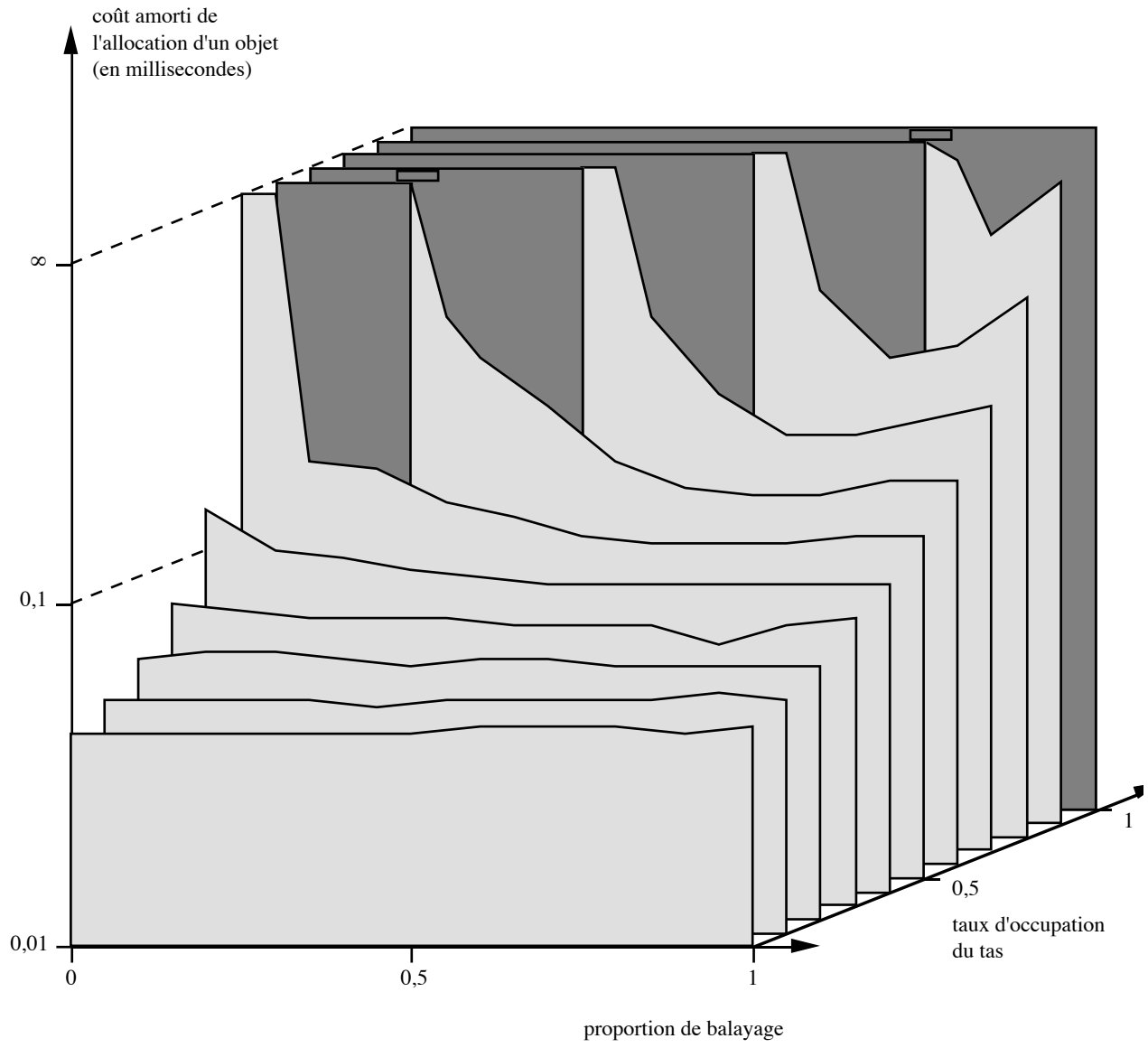


figure 15 : comportement du GC de Lang et Dupont

balayage n'est pas toujours plus coûteux que le GC à copie. On peut aussi décrire ce phénomène en disant que le coût en mémoire du GC à copie se traduit par un coût en temps quand le taux d'occupation du tas augmente.

Le GC de Lang et Dupont.

L'algorithme de Lang et Dupont a un degré de liberté supplémentaire, qu'il faut exploiter pour obtenir une performance optimale : c'est la proportion entre copie et balayage, que le GC peut choisir lui-même. La proportion idéale dépend évidemment du taux d'occupation du tas.

J'ai tracé le graphe qui donne le coût amorti de l'allocation en fonction du taux d'occupation du tas et de la proportion traitée en balayage (figure 15.)

On constate tout d'abord que le coût de l'allocation croît toujours avec le taux d'occpation du tas.

Lorsque le taux d'occupation est assez bas, la courbe est presque plate, ce qui s'explique par le faible coût du GC : celui-ci consomme seulement 10% du temps de calcul, ce qui fait que les variations du coût du GC sont difficiles à mesurer.

Je m'attendais cependant à une différence plus prononcée en faveur du GC à copie. L'exemple choisi (allocation dans deux listes distinctes) tend peut-être à sous-estimer la fragmentation de la zone traitée en balayage pour les faibles taux d'occupation.

Quand le taux d'occupation augmente, on observe un creux dans la courbe, qui indique la proportion idéale pour le GC, et le coût correspondant.

Il est très intéressant de remarquer que lorsqu'on fait varier les proportions, on n'obtient pas seulement des comportements intermédiaires entre "copie pure" et "balayage pur"; le mélange des deux est **plus efficace** que l'un ou l'autre. Cette efficacité accrue est due au compactage de la mémoire dans la zone copiée.

En projetant le minimum de chaque plan de la figure 15, on obtient la courbe du coût du GC de Lang et Dupont en fonction du taux d'occupation du tas. Cette courbe se trouve sur la figure 16, avec celles des GC à copie et à balayage. Cette courbe montre que le GC de Lang et Dupont est aussi

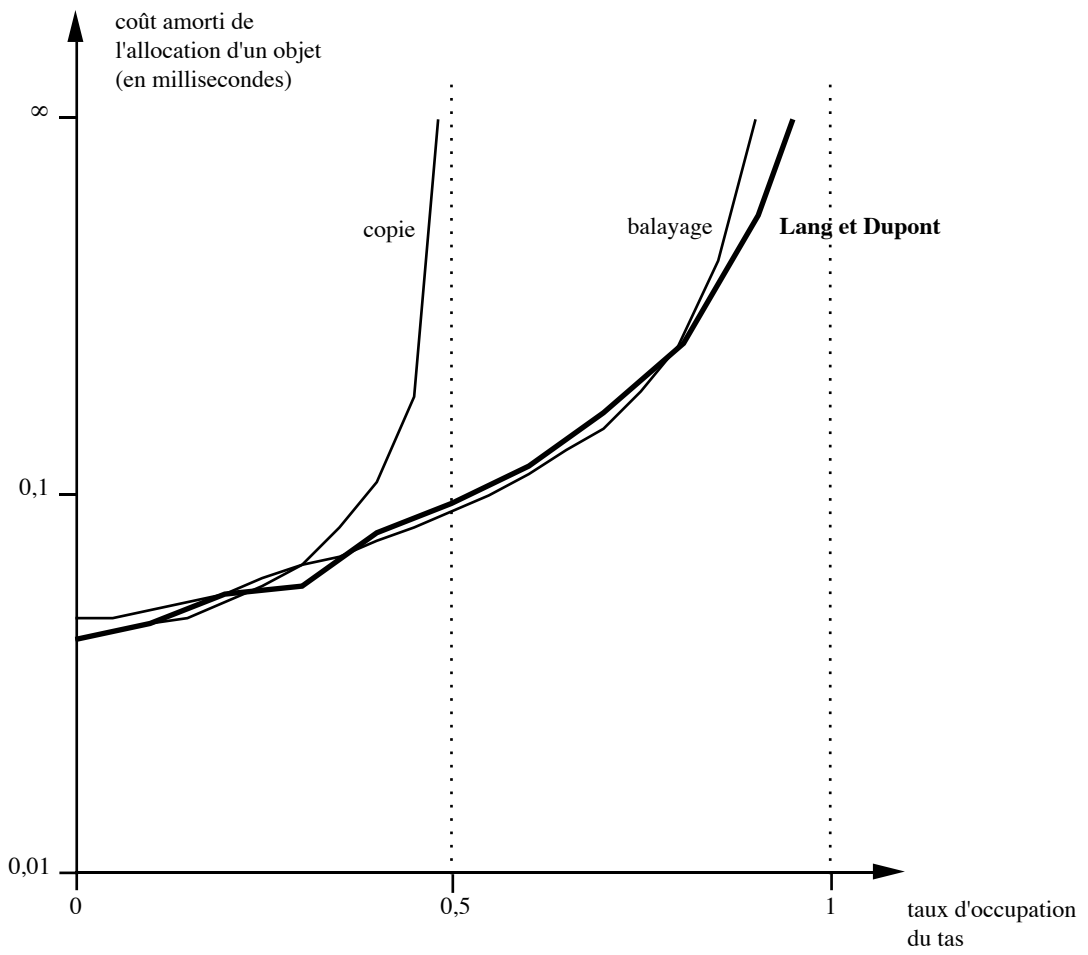


figure 16 : l'algorithme de Lang et Dupont comparé aux deux autres

efficace que le GC à copie quand celui-ci est utilisable, ou le GC à balayage dans les autres cas : l'algorithme de Lang et Dupont a donc tenu ses promesses.

Mémoire virtuelle.

Dans tout ce qui précède, je n'ai pas tenu compte d'un autre degré de liberté du GC : la taille du tas.

Il est évident qu'il faut la prendre aussi grande que possible, puisqu'alors le taux d'occupation du tas est minimal, ce qui constitue la condition optimale pour chacun des trois algorithmes. Cependant, il faut tenir compte d'une donnée importante : le coût de la mémoire virtuelle.

En effet, si on agrandit trop la taille du tas, le système d'exploitation fait intervenir la pagination de la mémoire, et une partie de celle-ci se trouve alors sur le disque, ce qui ralentit considérablement les accès. Malheureusement, ce ralentissement est difficilement mesurable sous UNIX, mais on peut donner un ordre de grandeur : pour une taille du tas inférieure à la moitié de la taille réelle, le coût est négligeable. Quand on a un tas aussi grand que la mémoire réelle, le programme d'exemple est ralenti d'un facteur 5 environ, et le ralentissement est d'autant plus fort si le GC fait beaucoup de copie.

Je propose donc l'heuristique suivante pour le pilotage du GC, en fonction du nombre de mots utiles dans la mémoire : maintenir un taux d'occupation de 25% tant que la taille du tas est inférieure à la moitié de la mémoire réelle, puis garder la même taille de tas jusqu'à ce que le taux d'occupation atteigne 60%, enfin agrandir le tas pour maintenir le taux d'occupation à 60%; la proportion de balayage étant réglée en fonction du taux d'occupation du tas. Il faut bien entendu faire varier les seuils en fonction de la machine, du langage, et même du programme.

On peut aussi imaginer de mesurer le coût de la gestion de la mémoire lors de chaque GC, et introduire une rétroaction sur la taille du tas et la proportion de balayage pour trouver par approximations successives les paramêtres idéaux.

Il est également possible de donner à l'utilisateur l'accès aux résultats des mesures effectuées par le GC, et de lui donner le contrôle des paramêtres de celui-ci, ce qui permet de régler le GC de la meilleure manière qui soit, en prévision du comportement futur du programme plutôt qu'en fonction de son comportement passé.

Générations

Je n'ai pas fait de mesures précises, mais j'ai pu constater que le mécanisme de générations apporte un gain important en temps de calcul et en confort d'utilisation, surtout lorsque la mémoire est pleine et les GC fréquents. En effet, le GC mineur filtre jusqu'à 97% des objets alloués, pour un temps de latence de l'ordre de 1/100 s. et un temps de calcul faible. Pour l'améliorer encore, on pourrait faire varier dynamiquement la taille de la zone mineure.

5. Conclusion

J'ai réalisé la première implémentation d'un GC de Lang et Dupont à générations; il est portable et efficace. Il me semble suffisament robuste pour être utilisé en vraie grandeur dans un système tel que CAML. Il ne me reste plus qu'à réaliser l'intégration du GC au système CAML, et le rendre incrémental, pour améliorer encore le confort d'utilisation.

Remerciements

Je tiens à remercier Bernard Lang, qui m'a fourni une grande partie de la bibliographie.

Je remercie Jean-Jacques Lévy qui m'a laissé les mains libres pendant toute la durée de mon stage, et qui est même allé jusqu'en Amérique du sud pour me laisser travailler tranquillement.

Je remercie aussi Xavier Leroy qui a pris un plaisir sadique à couvrir de remarques désobligeantes les précédentes version de ce document.

Je remercie enfin l'ensemble du projet Formel, pour CAML et son GC inefficace, qu'il est très intéressant d'améliorer.

Bibliographie

App 89 A. W. APPEL, *A runtime system*, brouillon, Princeton University, février 1989.

Bak 78 H. G. BAKER, List processing in real time on a serial computer, *Communications of the ACM*, Vol. 21, N°4, avril 1978, p. 280-294.

Ben 84 M. BEN-ARI, Algorithms for on-the-fly garbage collection, *ACM transactions on programming languages and systems*, vol 6, N° 3, juillet 1984, p. 333-344.

C&H 87 G. COUSINEAU, G. HUET, *The CAML Primer,* INRIA, 1987.

Car 86 L. CARDELLI. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and functionnal programming languages,* Springer-Verlag, 1986. Lecture Notes in Computer Science N° 242.

Cha 86 J. CHAILLOUX, M. DEVIN, F. DUPONT, J.-M. HULLOT, B. SERPETTE, J. VUILLEMIN, *Le_Lisp version 15.2, le manuel de référence*, INRIA, mai 1986.

Coh 81 J. COHEN, Garbage collection of linked data structures, *Computing surveys*, Vol 13, N° 3, septembre 1981, p. 341-367.

Daw 82 J. L. DAWSON, Improved effectiveness from a real time Lisp garbage collector, *Conference records of the 1982 ACM symposium on Lisp and functionnal programming*, août 1982, p. 159-167.

DeT 88 J. DETREVILLE, Experience with concurrent garbage collectors for Modula–2^{+}. septembre 1988. Non publié.

Dij 78 E.W. DIJKSTRA, L. LAMPORT, A.J. MARTIN, C.S. SCHOLTEN, E.F.M. STEFFENS, On-the-fly garbage collection : an exercise in cooperation, *Communications of the ACM*, Vol. 21, N° 11, novembre 1978, p. 967-975.

ELA 88 J. R. ELLIS, K. LI, A. W. APPEL, *Real-time concurrent collection on stock multiprocesors.*

FOR 89 P. WEIS, M.-V. APONTE, A. LAVILLE, M. MAUNY, A. SUÁREZ, *The CAML reference manual*, INRIA, 1989.

G&R 83 A.GOLDBERG, D.ROBSON, *Smalltalk-80 : the language and its implementation.* Addison-Wesley, Reading, Massachussets, 1983.

H&M 86 R. HARPER, K. MITCHELL, *Introduction to standard ML*, Laboratory for fundations of computer science, University of Edinburg, 1986.

Hal 85 R. H. HALSTEAD, Multilisp : a language for concurrent symbolic computation, *ACM transactions on programming languages and systems,* vol. 7, N° 4, octobre 1985, p. 501-538.

K&R 88 B. W. KERNIGHAN, D. M. RITCHIE, *The C programming language*, Prentice Hall, Englewood Cliffs, New Jersey, 2°édition, 1988.

Bibliographie

K&S 77a H. T. KUNG, S. W. SONG, *Performance analysis of a parallel garbage collection system*, Department of Computer Science report, Carnegie-Mellon University, 1977.

K&S 77b H.T. KUNG, S.W.SONG, An efficient garbage collection system and its correctness proof, *Proceedings of the IEEE 18th symposium on foundations of computer science,* Providence (R. I.) p. 120-131, octobre-novembre 1977.

Knu 68 D. E. KNUTH, *The art of computer programming, vol. 1 : fundamental algorithms,* Addison-Wesley, Reading, Massachussets, 1968.

L&D 88 B. LANG, F. DUPONT, Incremental incrementally compacting garbage collection, *Programming of future generation computers II,* Elsevier science publishers B.V. (North-Holland), 1988, p. 163-182; aussi dans : *SIGPLAN Notices*, Vol 22, N° 7, Juillet 87, pp. 253-263.

L&H 83 H. LIEBERMAN, C. HEWITT, A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM*, Vol. 26, N° 6, juin 83, p. 419-429.

N&R 87 S. C. NORTH, J. H. REPPY, Concurrent Garbage collection on stock hardware, *Proceedings of the 3rd conference on functionnal programming languages and computer architecture*, septembre 1987.

QBQ 89 C. QUEINNEC, B. BEAUDOING, J.-P. QUEILLE, Mark DURING sweep rather than mark THEN sweep, *PARLE 89,* Springer-Verlag, juin 1989, p. 224-237.

Spi 89 E. SPIR, *Etude et implantation d'un glaneur de cellules adaptatif pour LISP,* thèse de doctorat, Université de Paris VII, janvier 1989.

Ung 84 D. UNGAR, generation scavenging : a non-disruptive high performance storage reclamation algorithm, *Proceedings of the ACM SIGSOFT/SIGPLAN conference on practical programming environments*, p. 157-167, avril 1984.