THÈSE

présentée à

L'UNIVERSITÉ PARIS 7

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS 7

Spécialité : Informatique

par

Damien DOLIGEZ

Sujet de la thèse :

Conception, réalisation et certification d'un glaneur de cellules concurrent

Soutenue le 5 mai 1995 devant le jury composé de

MM.	**Guy COUSINEAU**	**Président**
	Gérard BERRY	**Rapporteurs**
	Joseph SIFAKIS	
	Albert BENVENISTE	**Examinateurs**
	Benjamin GOLDBERG	
Mme	**Thérèse HARDIN**	
MM.	**Leslie LAMPORT**	
	Jean-Jacques LÉVY	

version définitive
9 Septembre 1996
T

Remerciements

- M. Guy Cousineau, a bien voulu présider le jury. C'est aussi lui qui m'a initié aux joies des langages fonctionnels en général et de ML en particulier. C'est aussi grâce à lui que j'ai découvert que l'informatique est aussi une science. Je l'en remercie chaleureusement.

- MM. Gérard Berry et Joseph Sifakis ont eu la patience de lire cette thèse jusqu'au bout et ils ont accepté de rédiger les rapports. Ils ont droit à toute ma gratitude.

- MM. Albert Benveniste et Benjamin Goldberg ont bien voulu faire partie de mon jury. Qu'ils en soient remerciés.

- Je tiens particulièrement à exprimer ma gratitude envers Mme Thérèse Hardin : cet ouvrage n'aurait jamais vu le jour sans ses encouragements persistants et persuasifs, et sa relecture attentive d'un nombre considérable de versions préliminaires.

- M. Leslie Lamport a accepté de faire partie de mon jury. Il est aussi l'auteur du système LaTeX, qui a permis la réalisation matérielle de cette thèse, et dont l'emploi s'est parfois révélé "intéressant". Enfin, il a travaillé sur l'algorithme de base et il a inventé TLA. Étant juché sur les épaules d'un tel géant, je me dois de le remercier particulièrement.

- M. Jean-Jacques Lévy, en tant que directeur de thèse, m'a toujours laissé toute liberté sur le plan scientifique. Je l'en remercie vivement.

Je remercie aussi MM. Georges Gonthier et Xavier Leroy pour leur collaboration fructueuse autant qu'agréable, qui fut essentielle à la réalisation des travaux décrits dans cette thèse.

Je tiens aussi à remercier, pour des discussions courtes ou longues, mais toujours intéressantes : Pierre Weis, Michel Mauny, François Rouaix, Luc Maranget, Henri Laulhère, Christian Queinnec, Marc Shapiro, Emmanuel Chailloux, Doug Currie, Peter Sestoft, Greg Nelson, Dave MacQueen, Brad Chen et Daniel de Rauglaudre.

C'est M. Bernard Lang qui m'a initié au domaine du GC : dès le premier jour où j'ai mis les pieds à l'INRIA, il m'a fourni une partie de l'imposante bibliographie qu'il a glanée au fil des ans. Il a aussi réussi à me communiquer son enthousiasme pour ce sujet plutôt abstrait. Je l'en remercie vivement.

Enfin, je tiens à remercier Mmes Josy Baron, Ghislaine LeCorre, Sylvie Loubressac et Michelle Sarfati pour leur compétence, leur efficacité et leur bonne humeur.

Sommaire

Chapitre 1

Introduction

The proofs are extremely complex; it would be difficult to check them mechanically.
— Mordechai Ben-Ari [15]

La mémoire d'un ordinateur est composée de cases numérotées contenant chacune un octet. On peut grouper les octets contenus dans les cases pour former des nombres plus grands, et utiliser ces nombres pour représenter les numéros d'autres cases de la mémoire. Un numéro de case est appelé une *adresse*, et un groupe de cases qui contient une adresse est appelé un *pointeur*. On peut à nouveau grouper les pointeurs en *objets* et les objets en *structures de données*. C'est ainsi que l'on représente des choses complexes comme des vecteurs, des matrices, des listes, des arbres, des graphes ou des ensembles, dans les cases numérotées de la mémoire d'un ordinateur.

Dans un langage de programmation de haut niveau, le programmeur ne s'intéresse pas aux octets, aux adresses ou aux pointeurs : il manipule simplement les objets et il élabore des structures de données indépendamment des adresses. Il faut pourtant bien donner une adresse à chaque objet spécifié par le programmeur et y stocker la représentation de cet objet. C'est le travail d'un morceau de programme appelé le *gestionnaire de mémoire.*

Le gestionnaire de mémoire trouve une adresse libre pour stocker chaque nouvel objet demandé par le programme (c'est l'allocation) et il libère les adresses occupées par les objets devenus inutiles (c'est la désallocation). Un gestionnaire de mémoire *manuel* oblige le programme à lui signaler les objets inutiles, tandis qu'un gestionnaire de mémoire *automatique* détecte et désalloue de lui-même les objets inutiles.

La gestion manuelle de la mémoire est une source inépuisable d'erreurs faciles à commettre et difficiles à repérer : il suffit de désallouer par erreur un objet qui est encore utile, et le comportement du programme devient totalement erratique. La gestion automatique de la mémoire libère le programmeur de ces erreurs, et un langage de haut niveau ne se conçoit plus sans gestion automatique de la mémoire.

Cette thèse est consacrée à la conception, la preuve et l'implémentation d'un ges-

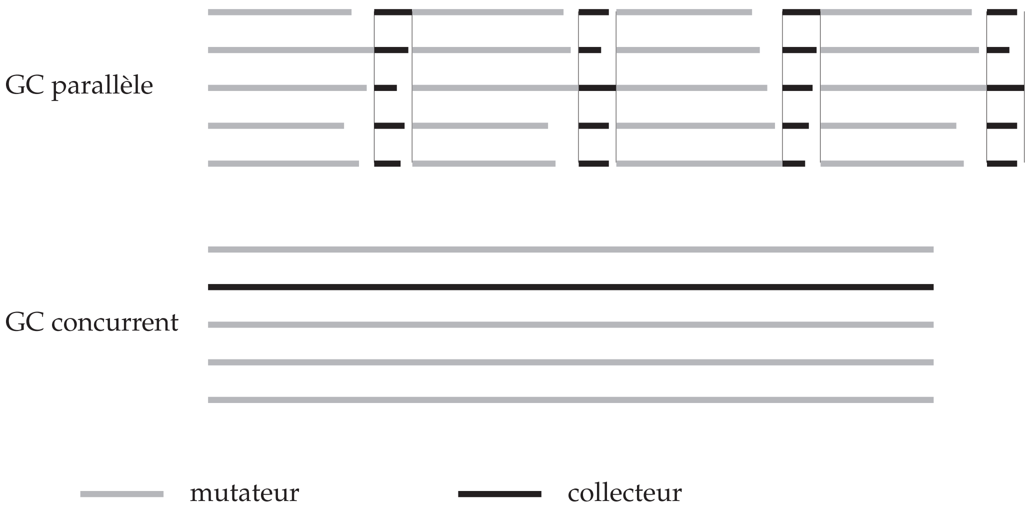


Figure 1.1: Comparaison des techniques de GC sur machines parallèles : chaque ligne horizontale représente le travail effectué par un processeur au cours du temps

tionnaire de mémoire automatique destiné à être utilisé avec des programmes écrits dans un langage de haut niveau autorisant l'exécution parallèle.

La méthode de gestion de mémoire la plus simple pour une machine parallèle consiste à adapter directement la technique utilisée pour les machines séquentielles : on arrête le programme, le temps d'effectuer un *glanage de cellules* (GC), c'est-à-dire de trouver tous les objets inutilisés et de les désallouer. Cette méthode est inadaptée car elle introduit des synchronisations parasites dans le programme : il faut arrêter tous les processus en même temps avant de déclencher le GC. Mais il est difficile d'arrêter tous les processus précisément au même moment, et ceux qui s'arrêtent en premier perdent du temps en attendant les autres. De plus, cette technique arrête le déroulement du programme pour un temps qui peut être assez long. Ce temps d'arrêt s'avère très gênant pour certaines applications.

La technique la plus intéressante pour la gestion de mémoire sur machines parallèles est le GC *concurrent* : le gestionnaire de mémoire travaille en même temps que les processus de l'utilisateur, avec très peu de synchronisation. On peut comparer cette technique et la précédente sur la figure 1.1. Les GC concurrents sont difficiles à mettre en œuvre, comme le montre le fossé existant entre théorie et pratique dans ce domaine : les algorithmes prouvés ne sont pas implémentés (car ils sont irréalistes) et les algorithmes implémentés ne sont pas prouvés (donc ils sont incorrects).

La première contribution de ce travail est de combler ce fossé en décrivant un algorithme de GC concurrent portable et efficace, et une preuve de correction de cet algorithme. L'algorithme est un GC concurrent à balayage inspiré de [33], mais qui tient compte de la présence de mémoires locales : registres, caches, piles, etc. Cet algorithme n'impose aucun surcoût aux opérations les plus courantes des mutateurs :

manipulations de la mémoire locale et lecture de la mémoire partagée ; il n'impose qu'un surcoût modéré (pas de synchronisation par verrous ou sémaphores) pour les autres opérations, sauf quand c'est inévitable : pour réserver de la mémoire prise dans la liste libre globale.

Dans cet algorithme, le collecteur doit prévenir les mutateurs du démarrage d'un nouveau cycle de GC. Il le fait par des "poignées de mains", qui nous permettent d'orienter cette synchronisation : c'est toujours le collecteur qui attend les mutateurs et jamais le contraire. De plus, un mutateur n'attend jamais un autre mutateur à cause du GC, donc nous n'introduisons pas de synchronisation "parasite" entre les mutateurs. Le résultat est que notre algorithme "dérange" extrêmement peu les mutateurs : le gestionnaire de mémoire fait son travail en retardant le moins possible le programme.

La preuve de notre algorithme utilise le formalisme TLA [59]. Nous prouvons un modèle formel de l'algorithme : c'est une version de l'algorithme écrite en TLA, dont nous avons bien sûr supprimé certains détails, mais nous avons eu à cœur de ne pas ignorer les détails de l'algorithme qui interfèrent avec la preuve (et qui sont donc des bogues potentiels). Par exemple les objets de taille variable posent des problèmes très subtils d'interférence avec le code de balayage : lorsqu'on alloue un objet dans un bloc libre de taille supérieure à cet objet, il faut diviser ce bloc en deux. Nous avons découvert qu'il faut que l'objet alloué soit placé dans les adresses hautes du bloc libre. Si on le place dans les adresses basses, le balayage risque de "rater" les objets alloués ultérieurement dans le reste du bloc libre. Ce type de problèmes justifie largement l'effort fourni pour prouver l'algorithme : une erreur de ce genre risque de passer inaperçue pendant tous les tests, puis de se déclencher brusquement (et systématiquement) chez un utilisateur à cause de circonstances particulières : charge de la machine, détails du fonctionnement du programme, etc. Le fait que ce problème concerne les objets de taille variable, qui sont souvent considérés comme un détail d'implémentation, justifie notre choix de prouver une version aussi détaillée que possible de l'algorithme, plutôt qu'une version abstraite plus facile à comprendre et à prouver, comme c'est l'habitude dans ce domaine.

Nous avons constaté que la preuve de programmes est un excellent outil de déboguage pour les programmes parallèles. Après avoir fait fonctionner le GC concurrent pendant quelque temps sans problèmes, nous avons formalisé l'algorithme, ce qui nous a permis de trouver deux erreurs. Nous avons ensuite prouvé l'algorithme, ce qui a révélé une dernière erreur. Enfin, Georges Gonthier a écrit cette preuve sous une forme vérifiable par machine, et il a trouvé une omission importante dans la preuve "informelle" (qui ne se traduit heureusement pas par un bogue supplémentaire de l'algorithme).

En complément de la conception et de la preuve de notre GC concurrent, nous avons également attaqué les problèmes propres aux langages à fort taux d'allocation, en particulier le langage ML. Les programmes ML utilisent beaucoup d'objets, dont la plupart deviennent presque immédiatement inutiles. Cette façon d'utiliser la mémoire est incompatible avec l'utilisation simple d'un GC à balayage car celui-ci ne peut pas

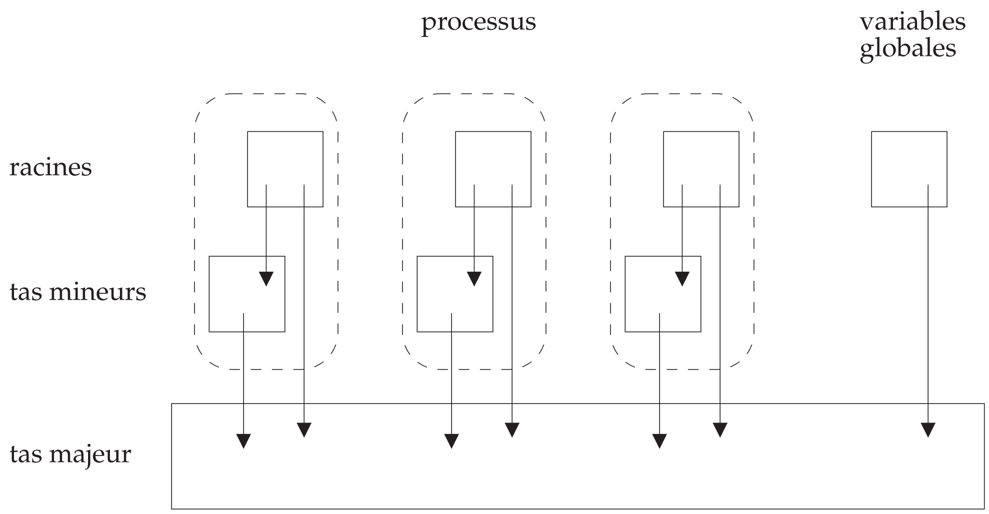


Figure 1.2: Organisation de la mémoire pour le GC concurrent à générations

désallouer les objets aussi vite que le programme les alloue. La solution à ce problème réside dans l'utilisation d'un GC à *générations*, qui concentre ses efforts sur la partie de la mémoire qui contient le plus d'objets inutiles. Or les GC à générations efficaces sont des GC à copie (au moins pour la zone mineure), mais il n'y a pas de moyen connu de faire un GC concurrent à copie qui soit portable et efficace.

La deuxième contribution de ce travail est un système de générations qui résout ce dilemme en utilisant un GC mineur à copie non concurrent et un GC majeur concurrent sans copie. Dans notre implémentation, le GC majeur est l'algorithme concurrent décrit ci-dessus, mais notre système de générations pourrait s'adapter à d'autres GC majeurs. Dans ce système, chaque processus est muni d'un tas mineur, dans lequel il alloue ses objets et dans lequel il effectue lui-même les GC mineurs (en stoppant momentanément ses calculs). Cette organisation de la mémoire est illustrée par la figure 1.2. Les GC mineurs ne sont pas synchronisés : un processus peut effectuer un GC mineur à tout moment sans prévenir les autres. La répartition du temps de calcul que nous obtenons avec ce système est illustrée par la figure 1.3 : les GC mineurs se déclenchent de façon imprévisible, mais chaque GC n'interrompt qu'un processus ; le GC majeur est parfois obligé d'attendre, mais les mutateurs n'attendent jamais.

Pour que ce système fonctionne, il faut que le tas mineur de chaque processus soit privé, c'est-à-dire que les autres processus n'y aient pas accès. Pour s'en assurer, il suffit d'interdire la création de pointeurs du tas majeur vers les tas mineurs. De tels pointeurs ne peuvent être créés que par la primitive d'affectation ; nous modifions donc celle-ci pour qu'elle recopie dans le tas majeur les objets mineurs pointés par un objet majeur au lieu de créer un pointeur interdit. Cette copie aboutit à une duplication temporaire de l'objet créé, qui n'est pas gênante car le langage ML ne fournit pas de fonction pour tester l'égalité "physique" de deux objets.

Il faut cependant éviter de copier les objets *mutables*, c'est-à-dire ceux dont le pro-

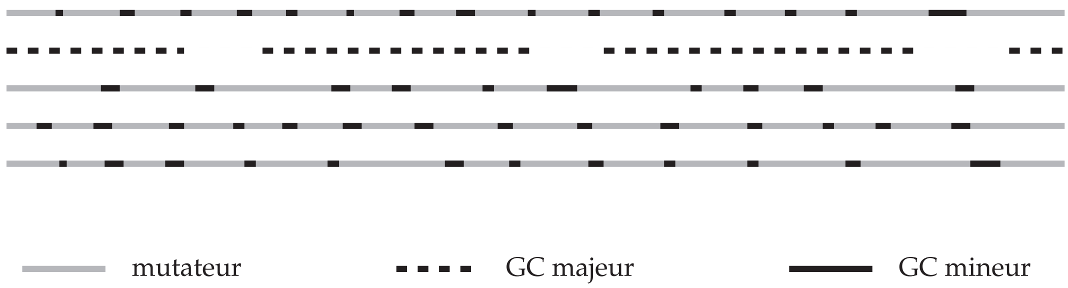


Figure 1.3: Répartition du temps de calcul entre mutateurs, GC mineurs et GC majeur

gramme peut changer la valeur. En effet, si on change la valeur d'une copie sans changer l'autre, la sémantique du langage n'est pas respectée. Notre solution est d'utiliser les informations de typage de ML, qui nous permettent d'allouer directement dans le tas majeur ces objets mutables, ce qui garantit qu'ils ne seront pas copiés. C'est une application originale et inattendue du typage statique au domaine de la gestion de la mémoire.

Plan

L'organisation de cette thèse est la suivante :

Le chapitre 2 explique les techniques de base de gestion de la mémoire. Il décrit non seulement les techniques utilisées dans la suite, mais aussi les autres techniques disponibles. La plupart des concepts utilisés dans les autres chapitres sont définis ici (et indiqués dans l'index).

Le chapitre 3 décrit une expérience préliminaire effectuée avec l'algorithme de GC de Lang et Dupont et un mécanisme de générations. Les leçons tirées de cette expérience ont une certaine importance pour la suite, notamment en ce qui concerne les générations.

Dans le chapitre 4, nous exposons les difficultés rencontrées dans la conception du GC concurrent et nous donnons une description informelle de ce GC (dans une version légèrement simplifiée).

Le chapitre 5 présente la version formelle de l'algorithme concurrent, sous forme de formules TLA. Celles-ci sont bien entendu largement commentées.

Le chapitre 6 donne, encore une fois sous forme de formules commentées, toutes les définitions préliminaires nécessaires à la preuve. Faute de place et de temps, la preuve elle-même n'est pas donnée dans cette thèse (la version "mécanisée" de cette preuve sera publiée par ailleurs). Le lecteur motivé (et spécialiste de TLA) devrait trouver dans ce chapitre suffisamment d'éléments pour écrire la preuve lui-même.

Dans le chapitre 7, nous exposons le fonctionnement de notre système de générations pour machines parallèles et nous décrivons les implémentations réalisées, l'une concurrente pour machines parallèles et l'autre incrémentale pour machines séquentielles.

Enfin, l'annexe A reprend, sans les commentaires, toutes les formules présentées dans les chapitres 5 et 6 et l'annexe B donne une idée de la taille de la preuve dans sa version mécanisée.

Chapitre 2

Techniques de base

2.1 Définitions

2.1.1 Gestion de la mémoire dans les langages de programmation

L'architecture de Von Neumann est caractérisée par le fait que la mémoire contient à la fois le programme et les données qu'il manipule. La mémoire n'est rien d'autre qu'un tableau d'octets, repérés par leurs adresses. Au cours de son exécution, le programme est amené à lire et à écrire des données dans la mémoire. Pour cela, il lui faut interpréter des suites d'octets comme des entiers, des chaînes de caractères, ou des valeurs structurées telles des listes, des graphes, etc. L'utilisation de structures de données complexes facilite le codage des algorithmes mais complique la gestion de la mémoire. Les langages de haut niveau sont caractérisés non seulement par la complexité des algorithmes qu'ils permettent d'implémenter, mais aussi par la richesse des structures de données qu'ils permettent de manipuler. La complexité de ces structures exige le développement d'outils sophistiqués pour la gestion de la mémoire.

Dans les premiers programmes, écrits en langage machine, le programmeur attribuait à chaque donnée son adresse précise dans la mémoire (généralement écrite directement en hexadécimal) au moment d'écrire le programme. Dans le premier langage de programmation (FORTRAN), le programmeur désigne les adresses par des noms symboliques, mais il doit toujours attribuer une adresse à chaque donnée au cours de l'écriture du programme. C'est ce qu'on appelle l'*allocation statique* de la mémoire.

Cette méthode fort utile se révèle insuffisante dès qu'on veut manipuler des tableaux dont la taille n'est connue qu'à l'exécution, des listes, etc. : la manipulation de listes en FORTRAN est le problème qui a motivé la création de Lisp [73]. D'autre part, la mémoire est aussi utilisée pour stocker des résultats intermédiaires, dont la durée d'utilisation n'est connue qu'à l'exécution. Pour résoudre ce problème, on utilise un mécanisme d'*allocation dynamique* : un *gestionnaire de mémoire* (GM) fournit des primitives d'allocation et de désallocation que le programme appelle au cours de son exécution. L'allocation dynamique est utilisée par deux familles de langages, qui diffèrent par la façon dont la désallocation est gérée.

La première famille (Pascal, C, ...) utilise un mécanisme de désallocation explicite : lorsqu'un programme a alloué une zone de mémoire, cette zone ne sera désallouée — et donc considérée comme à nouveau libre par le GM — que si le programme le demande par un appel à la fonction de désallocation.

La désallocation explicite est d'utilisation délicate : il faut désallouer les zones de mémoire qui deviennent inutilisées, mais ni trop tôt ni trop tard. Si on oublie de désallouer les zones de mémoire devenues inutiles, le gestionnaire de mémoire ne peut pas les recycler, et le programme consomme de plus en plus de mémoire. Lorsqu'il a rempli toute la mémoire disponible, il ne peut plus travailler. Ce défaut de désallocation s'appelle une *fuite de mémoire*. Si en revanche on désalloue une zone de mémoire dont on a encore besoin, le gestionnaire de mémoire la recycle. Le programme stocke alors une nouvelle donnée à la place d'une donnée encore utile, d'où un résultat complètement ou partiellement faux. Ce problème s'appelle le problème du *pointeur fou*.

La deuxième famille de langages avec allocation dynamique (Lisp, ML, SmallTalk, Modula 3, ...) fournit un système de désallocation implicite. Le GM est alors constitué d'une fonction d'allocation et d'un *glaneur de cellules* (GC, ou *collecteur*). Le programme est alors appelé *mutateur*, par opposition au collecteur (cette terminologie a été introduite par [33]). Il continue d'allouer explicitement la mémoire dont il a besoin en appelant la fonction d'allocation, mais il n'y a plus de fonction de désallocation. Le GC détermine quelles parties de la mémoire ne sont plus utilisées par le mutateur et il les recycle automatiquement. Cette gestion automatique de la mémoire est un outil très utile pour le programmeur car elle supprime à la fois les fuites de mémoire et les pointeurs fous, qui sont des erreurs faciles à commettre et difficiles à repérer et à corriger.

2.1.2 Organisation de la mémoire

On appelle *objet*[1] une zone de mémoire allouée par la primitive d'allocation. Le mutateur référence les objets par des *pointeurs* (un pointeur vers un objet est une représentation de l'adresse de cet objet), qui sont stockés dans les variables du programme et dans d'autres objets. Les variables du programme sont appelées les *racines*. Les objets pointés par les racines sont appelés les *objets-racines*. Si un objet A contient un pointeur vers un objet B, on dit que B est un *fils* de A. B est un *descendant* de A si B est un fils de fils ... de A. Un objet est dit *vivant* ou *accessible* si c'est un descendant d'un objet-racine. Un objet qui n'est pas vivant est dit *mort* ou *inaccessible*.

Si un objet est inaccessible, le mutateur a perdu son adresse, et il ne peut donc plus l'utiliser. Le travail du GC est de repérer les objets inaccessibles et de les désallouer (les objets inaccessibles sont inaccessibles au mutateur, mais le collecteur garde un moyen de les référencer). On appelle *tas* la partie de la mémoire gérée par le GC. Le reste de la mémoire peut contenir des objets alloués statiquement et des objets à désallocation explicite. En particulier, la plupart des systèmes utilisent une pile pour les variables

[1] Il ne s'agit pas d'objets au sens des langages à objets, même si les langages à objets utilisent généralement des objets du GM pour représenter leurs objets.

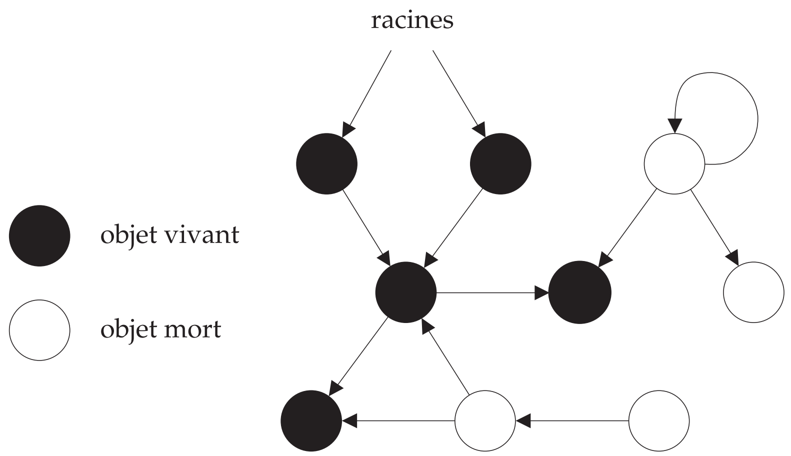


Figure 2.1: Le graphe mémoire

locales, les adresses de retour des sous-programmes et certains résultats intermédiaires. La plupart des racines se trouvent donc dans la pile sous forme de variables locales.

Il est utile de considérer une vision abstraite du tas, le *graphe mémoire*. Le tas est un graphe dont les nœuds sont les objets et les flèches sont les pointeurs (figure 2.1). La description des algorithmes de GC se fait généralement en termes de graphe mémoire : le mutateur modifie ce graphe en ajoutant des nœuds (en allouant des objets) et en changeant les flèches (en changeant les pointeurs contenus dans les objets). Cette dernière opération s'appelle une *affectation* ou *mutation* (d'où le nom de "mutateur"). Le collecteur identifie les nœuds inaccessibles et il les retire du graphe (en les désallouant).

2.2 Contraintes d'implémentation du GC

Cette section détaille les principaux facteurs qui ont une influence sur le choix d'une stratégie de GC pour un système particulier. Il n'y a pas de GC universel : il faut s'adapter au langage, à la machine, au système d'exploitation, au compilateur, aux programmes considérés, etc.

Familles de langages

On distingue plusieurs familles de langages, selon qu'ils disposent ou non de l'allocation dynamique et de la désallocation implicite, selon leur *taux d'allocation* (la fréquence d'appel à la fonction d'allocation), et leur *taux de mutation* (la fréquence des opérations d'affectation).

- FORTRAN n'a pas d'allocation dynamique ni de pointeurs. Il n'a donc pas besoin de GC.

- Pascal, C et C++ ont une allocation dynamique avec désallocation explicite. On peut remplacer la désallocation explicite par un GC, mais c'est assez rare (sauf pour C++). Leur taux d'allocation est faible et leur taux de mutation élevé. En effet, la syntaxe de ces langages et la désallocation explicite découragent l'utilisation de l'allocation dynamique et encouragent la réutilisation de la mémoire déjà allouée en remplaçant par affectation le contenu des objets.

- Lisp et Scheme ont la désallocation implicite. Le langage est fait pour que le programmeur contrôle finement l'allocation de mémoire, mais il facilite l'allocation dynamique, d'où un taux d'allocation relativement élevé. L'utilisation des fonctions d'affectation est moins courante qu'en C, ce qui donne un taux de mutation faible.

- ML a un taux d'allocation encore plus élevé que Lisp car les compilateurs allouent généralement de la mémoire pour implémenter certaines constructions du langage [5], et le langage permet de construire aisément des structures complexes par allocation dynamique. En revanche, le taux de mutation est très faible, car le système de types restreint l'affectation aux objets déclarés mutables par le programmeur.

- Les langages paresseux comme Haskell et Lazy-ML ont en général des taux d'allocation et de mutation très élevés : l'opération de mutation n'est pas disponible dans le langage, mais elle est utilisée par le compilateur sur les structures de données qui implémentent les calculs paresseux (c'est à dire presque toutes les structures de données).

Il faut noter que la différence (du point de vue du taux d'allocation) entre Lisp et ML ne se situe pas tellement dans le langage lui-même, mais plutôt dans les techniques de compilation et le style de programmation normalement utilisé. De plus, il est possible de programmer en Lisp sans prêter attention à l'allocation, ce qui donne des programmes proches de ML ; et beaucoup de compilateurs ML modernes essayent de réduire leur taux d'allocation, ce qui les rapproche de Lisp.

Machines multi-processeurs, programmes multi-tâches

Certaines techniques de GC peuvent s'adapter pour fonctionner sur des machines ayant au moins deux processeurs avec mémoire partagée : le mutateur et le collecteur s'exécutent simultanément, chacun sur son processeur, et ils travaillent en même temps sur le même graphe mémoire. Sur les machines multi-processeurs à mémoire partagée, on peut aussi faire fonctionner des programmes parallèles (avec plusieurs mutateurs), avec un ou plusieurs collecteurs qui récupèrent la mémoire. Les mêmes techniques serviront aussi pour les programmes multi-tâches sur les machines à un seul processeur.

Dans le cas des machines multi-processeurs à mémoire distribuée, on utilise des algorithmes de GC *réparti*, qui sont très différents des algorithmes pour machines séquentielles ou à mémoire partagée. Ces algorithmes de GC réparti servent aussi pour les systèmes distribués (où un programme s'exécute en parallèle sur un ensemble de machines connectées par un réseau). Nous ne traiterons pas le problème du GC réparti dans cette thèse. Il est abordé par exemple dans [16, 42, 51, 52, 63, 64, 71, 80, 3, 83, 84, 85, 92].

Primitives du système d'exploitation

Certains algorithmes de GC tirent profit de fonctions spéciales du système d'exploitation, notamment les fonctions de contrôle de la lecture et de l'écriture de la mémoire, pour déplacer des objets dans la mémoire à l'insu du mutateur. Ces fonctions de contrôle sont généralement fournies par le processeur et utilisées par le système d'exploitation pour ses besoins propres. Elles sont considérées comme des fonctions de bas niveau, elles sont peu répandues et leur fonctionnement varie d'un système à l'autre [8]. Les GC qui les utilisent sont donc peu portables, et leur efficacité varie beaucoup d'une machine à l'autre.

Programmes temps-réel ou interactifs

Le graphe mémoire est une structure de données commune au collecteur et au mutateur. Pour éviter les problèmes d'accès simultanés, les algorithmes simples de GC arrêtent le fonctionnement du mutateur le temps de faire tourner le collecteur, ce qui se traduit par une pause dans le déroulement du programme. La durée de cette pause est appelée le *temps de latence* du GC. Les programmes temps-réel exigent des temps de latence bornés (garantis courts), et les programmes interactifs supportent mal des temps de latence trop longs car l'utilisateur a l'impression que la machine est bloquée pendant le travail du collecteur. Les contraintes sur le temps de latence sont importantes dans le choix d'un algorithme de GC.

Interface entre le mutateur et le collecteur

Le graphe mémoire étant une structure de données partagée par le mutateur et le collecteur, sa représentation concrète est un "contrat" entre ces deux programmes. Le collecteur est généralement associé à un environnement de programmation et son code est partagé par de nombreux programmes. Il impose donc certaines contraintes sur l'organisation de la mémoire, que le mutateur doit respecter.

Le contrat le plus courant vise à garantir :

- que le collecteur connaît l'ensemble des racines (qui est en général contenu dans les variables globales, la pile et les registres),
- que le collecteur peut trouver tous les pointeurs contenus dans un objet donné,
- et que les pointeurs vers un objet pointent tous vers le début de cet objet. On appelle *pointeur infixe* un pointeur vers l'intérieur d'un objet. Les pointeurs infixes sont donc interdits dans le graphe mémoire.

La façon la plus simple de respecter ce contrat consiste à coordonner le compilateur et le GC. C'est le compilateur qui choisit la représentation (dans le tas et dans la pile) des données manipulées par le programme. Il lui suffit d'utiliser une représentation qui respecte les contraintes imposées par le GC. Il s'assure ainsi que le mutateur qu'il compile est compatible avec le collecteur.

Il est aussi possible d'utiliser un compilateur préexistant, qui n'est pas prévu pour coopérer avec un GC. Dans ce cas, on est obligé d'utiliser un algorithme de GC qui s'accommode d'un contrat beaucoup plus faible.

2.3 Techniques de GC

Cette section présente quelques techniques de base de GC. L'art de la conception de GC se résume bien souvent à trouver une combinaison de ces techniques de base qui s'adapte bien aux contraintes d'implémentation. On trouve dans [24, 98] une liste plus complète des techniques disponibles.

2.3.1 Comptes de références

La méthode la plus simple pour désallouer les objets inaccessibles consiste à associer à chaque objet un entier, appelé *compte de références*, qui est le nombre de pointeurs vers cet objet [19, 30, 50, 72, 100, 101]. En général on stocke cet entier dans la mémoire juste à côté de l'objet lui-même. Le mutateur met à jour ce compte à chaque fois qu'il crée ou supprime un pointeur. À l'allocation de l'objet, la fonction d'allocation initialise le compte à 1 avant de retourner un pointeur vers l'objet. Quand il duplique un pointeur le mutateur doit incrémenter le compte de l'objet correspondant, et quand il supprime un pointeur il doit décrémenter le compte. Si le compte tombe à zéro, l'objet n'est plus accessible et il faut le désallouer, en n'oubliant pas de décrémenter les comptes de tous ses fils, puisque les pointeurs qu'il contenait disparaissent.

Avantages et inconvénients

Le seul avantage du GC à comptes de références est qu'il désalloue les objets aussitôt qu'ils deviennent inaccessibles. Ainsi le programme n'utilise que la quantité de mémoire strictement nécessaire, et le travail de désallocation est généralement bien réparti entre les calculs du mutateur, ce qui donne des temps de latence faibles.

Les inconvénients sont au nombre de trois : le temps de calcul, la taille du code et les fuites de mémoire sur les structures circulaires.

Le coût en temps de calcul est très élevé, surtout pour les programmes qui effectuent beaucoup de manipulations de pointeurs. Une affectation qui remplace un pointeur par un autre dans un objet doit, en plus d'écrire le nouveau pointeur à la place de l'ancien, lire le compte de l'ancien objet, le décrémenter, écrire sa nouvelle valeur, la tester (et désallouer l'objet si elle est nulle), lire le compte du nouvel objet, l'incrémenter et écrire sa nouvelle valeur. De plus le programme doit aussi mettre à jour les comptes lors de toutes ses manipulations de variables (les racines).

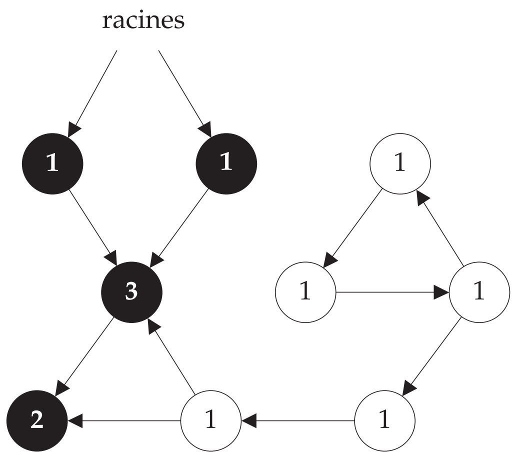


Figure 2.2: Comptes de références : les objets morts forment une structure circulaire qui ne sera pas désallouée.

La taille du code augmente elle aussi beaucoup, puisqu'une opération très courante qui demande normalement une seule instruction (écrire un pointeur dans la mémoire) exige maintenant une demi-douzaine d'instructions au minimum.

Enfin, les GC à comptes de références ne désallouent pas les structures circulaires. On appelle *structure circulaire* un cycle du graphe mémoire : un objet qui est pointé par un de ses descendants. Ces structures sont très utiles dans certains programmes, et le GC à comptes de références ne peut pas détecter qu'elles deviennent inaccessibles, puisqu'il reste toujours au moins un pointeur vers chaque objet (figure 2.2).

2.3.2 GC à balayage

Dans un GC à balayage (*mark and sweep*) le collecteur effectue un parcours complet du graphe mémoire, en coloriant les objets parcourus [9, 10, 56, 55, 73, 81, 89, 104]. Quand ce parcours (le *marquage*) est fini, on sait alors que les objets coloriés sont accessibles, et que tous les autres sont inaccessibles. Dans la deuxième phase (le *balayage*), le collecteur examine tour à tour tous les objets du tas et il désalloue ceux qui ne sont pas coloriés. L'ensemble des deux phases (marquage et balayage) est appelé un *cycle de GC* ou simplement un GC.

Marquage à deux couleurs

Dans sa variante la plus simple, le marquage travaille sur une variable booléenne associée à chaque objet : c'est sa couleur, qui peut être blanc ou noir. Au début de la phase de marquage, tous les objets sont blancs. Le GC noircit les objets-racines, puis il utilise les couleurs pour parcourir le graphe : il cherche un objet noir qui pointe vers un objet blanc ; celui-ci est un fils d'un objet accessible, il est donc accessible et le GC le noircit. Lorsqu'il n'y a plus de pointeur d'un noir vers un blanc, le marquage est fini.

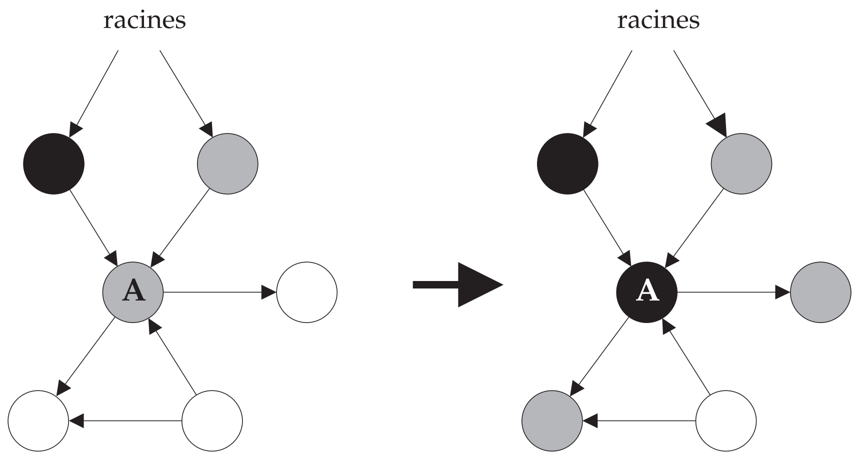


Figure 2.3: Marquage à trois couleurs : parcours de l'objet A

Pour trouver un pointeur d'un objet noir vers un blanc, on peut examiner tous les objets du tas, ce qui est très lent, surtout à la fin du marquage quand il reste peu de pointeurs à trouver. On peut aussi remarquer qu'un pointeur d'un noir vers un blanc ne peut apparaître que lorsqu'on noircit un objet blanc (puisque le GC ne crée pas de nouveau pointeur). Il suffit donc quand on noircit un objet de parcourir ses fils blancs (en les noircissant et en parcourant leurs fils et ainsi de suite) et on n'a plus à chercher les pointeurs des noirs vers les blancs. Cela se fait par un parcours récursif (en profondeur d'abord) du graphe.

Marquage à trois couleurs

Le parcours récursif du graphe a l'inconvénient d'utiliser de la mémoire. En effet, il faut se souvenir de l'objet en cours de parcours, le temps de parcourir chaque fils, pour pouvoir passer au suivant lorsque le parcours du fils est terminé. La quantité de mémoire utilisée par le marquage dépend de la forme du graphe mémoire, donc des structures de données utilisées par le programme. Or on déclenche le GC pour récupérer de la mémoire quand il n'y en a plus, donc il ne doit pas en utiliser trop lui-même. L'idéal serait que le GC utilise une quantité fixe de mémoire préallouée, pour pouvoir travailler quand il n'y a plus de mémoire disponible.

Pour obtenir ce résultat, on peut utiliser trois couleurs au lieu de deux : noir, gris et blanc. On utilise le gris pour les objets en cours de parcours : les objets qu'on sait vivants mais dont on n'a pas encore marqué les fils ; et le noir pour les objets (vivants) dont on a déjà marqué les fils. On commence par *griser* (colorier en gris s'ils sont encore blancs) les objets-racines. Le parcours du graphe se réduit alors à choisir un objet gris quelconque, griser ses fils et le noircir (figure 2.3). Le parcours du graphe se fait donc dans un ordre arbitraire.

Lorsqu'il n'y a plus d'objet gris, le marquage est fini, tous les objets accessibles sont noirs et tous les objets inaccessibles sont blancs. le balayage désalloue les objets blancs

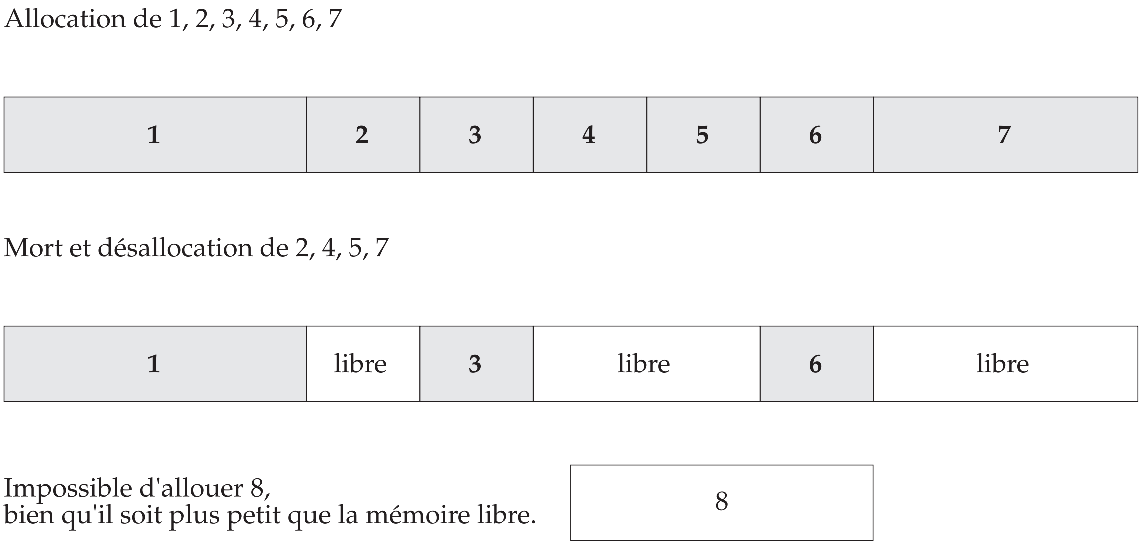


Figure 2.4: Fragmentation de la mémoire

et il blanchit les objets noirs pour préparer le prochain cycle.

La principale différence entre le marquage à deux couleurs non récursif et le marquage à trois couleurs est qu'il est plus facile de détecter un objet gris (il suffit d'examiner sa couleur) qu'un objet noir qui pointe vers un blanc (il faut examiner sa couleur et celle de tous ses fils).

Avantages et inconvénients

Le GC à balayage ne déplace pas les objets, ce qui est à la fois un avantage et un inconvénient.

Comme il ne déplace pas les objets, il n'a pas besoin de changer les pointeurs contenus dans les objets. Il est donc relativement facile à rendre concurrent ou incrémental (c'est-à-dire de faire travailler le collecteur en même temps que le mutateur). En effet, le mutateur travaille sur le contenu des objets sans changer les couleurs, tandis que le collecteur travaille sur les couleurs sans changer le contenu des objets.

Comme il ne déplace pas les objets, le GC à balayage ne *compacte* pas la mémoire : il ne peut pas déplacer un objet vivant situé entre deux zones libres pour obtenir une seule zone libre plus grande. Supposons que le programme alloue des objets de durées de vie différentes (ce qui est presque toujours le cas, sinon l'intérêt de la désallocation implicite est assez réduit). Lorsque les objets à courte durée de vie sont désalloués, on obtient des zones libres séparées par des objets encore vivants. Cela s'appelle la *fragmentation* de la mémoire libre.

Pour allouer un nouvel objet, il faut donc chercher une zone libre de taille suffisante, ce qui complique l'allocation. Il peut aussi arriver qu'il n'y ait pas de zone libre assez grande, même s'il reste beaucoup de mémoire libre, donc la fragmentation diminue la

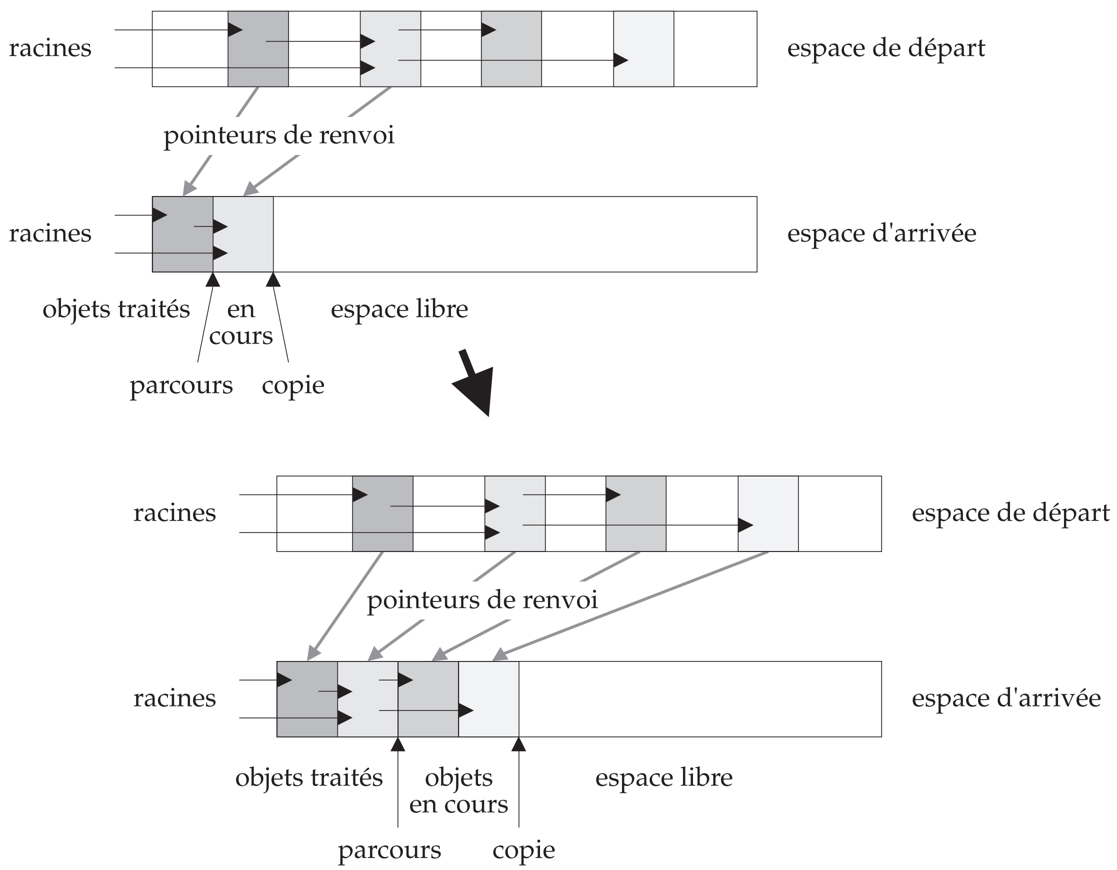


Figure 2.5: Principe du GC à copie

quantité de mémoire utilisable en pratique (figure 2.4).

2.3.3 GC à copie

Dans un GC à copie (*stop and copy* [4, 23, 34, 40, 41, 70, 94]), la mémoire est divisée en deux zones de tailles égales. L'une d'elles (l'*espace d'arrivée*) est réservée au GC et elle reste vide pendant que le mutateur travaille. Celui-ci alloue dans l'autre zone (l'*espace de départ*) jusqu'à ce qu'elle soit pleine. On arrête alors le mutateur et le collecteur parcourt le graphe mémoire en copiant chaque objet vivant dans l'espace d'arrivée. Il faut bien sûr mettre à jour tous les pointeurs puisqu'on déplace les objets (figure 2.5). Quand tous les objets vivants sont copiés, on inverse les rôles des espaces de départ et d'arrivée.

Le collecteur utilise deux pointeurs (le pointeur de parcours et le pointeur de copie) pour séparer l'espace d'arrivée en trois zones : les objet traités (avant le pointeur de parcours), les objets en cours (entre les deux pointeurs) et l'espace libre (après le pointeur de copie). Au début du cycle de GC, les deux pointeurs sont au début de l'espace d'arrivée, qui est donc entièrement libre. A chaque fois qu'il copie un objet, le collecteur le place au début de l'espace libre, puis il avance le pointeur de copie pour que cet objet soit dans les objets en cours.

Le collecteur commence par copier les objets-racines, puis il répète l'opération suivante : prendre le premier objet en cours, le placer dans les objets traités (en avançant le pointeur de parcours) et copier tous ses fils. Lorsqu'il n'y a plus d'objets en cours tous les objets vivants ont été copiés ; le cycle de GC est donc fini.

Pour éviter de copier plusieurs fois le même objet, on marque les objets copiés et on leur associe un *pointeur de renvoi* qui donne l'adresse de la copie. Lorsque le GC trouve un deuxième pointeur sur un objet déjà copié, il peut alors le mettre à jour en utilisant directement la copie. Le pointeur de renvoi ne fait pas partie du graphe mémoire : il n'est utilisé que par le GC et il n'existe que pendant le travail du GC. Il est généralement stocké dans l'espace de départ à la place de l'objet copié.

Avantages et inconvénients

Le GC à copie a deux gros avantages : la vitesse et le compactage. La copie s'avère plus rapide que le marquage, et le temps de GC est seulement proportionnel à la taille totale des objets accessibles (contre la taille totale du tas pour le GC à balayage). Le GC à copie recolle les objets vivants au début de l'espace d'arrivée, et la mémoire libre est donc d'un seul tenant, ce qui facilite l'allocation.

Le plus grave inconvénient du GC à copie est bien sûr qu'il se réserve la moitié de la mémoire. Il est aussi très difficile à rendre parallèle, concurrent ou incrémental. En effet, il déplace les objets et change les pointeurs que le mutateur utilise pour travailler. Il est difficile de laisser le mutateur travailler sur le graphe mémoire pendant la copie.

2.3.4 GC à générations

Le principe des générations [4, 27, 69, 90, 91, 97, 99] peut s'appliquer à un GC à balayage comme à un GC à copie. Il permet de gagner du temps en n'examinant que des objets qui ont une forte probabilité d'être morts. Le principe est le suivant :

L'ensemble des objets est divisé en deux parties, les objets *jeunes* et les objets *vieux* Les objets nouvellement alloués commencent par être jeunes. Un cycle de GC *mineur* examine les objets jeunes, désalloue ceux qui sont morts et fait passer les jeunes vivants dans l'ensemble des vieux. Un cycle de GC *majeur* examine tous les objets normalement et désalloue les objets morts. On appelle le *tas mineur* l'ensemble des objets jeunes et le *tas majeur* l'ensemble des objets vieux.

Le GC mineur n'a pas besoin de parcourir tout le graphe mémoire. En effet, les objets jeunes vivants sont accessibles à partir des racines ou à partir de pointeurs contenus dans les objets vieux. Supposons qu'un vieux pointe vers un jeune. Le jeune a été alloué plus récemment que le vieux, donc le pointeur a du être stocké dans l'objet vieux après son allocation. Dans certains langages, cette opération d'affectation dans un objet déjà alloué est rare[2] et il est facile de tenir à jour la liste des pointeurs des

[2] La plupart des objets sont donc plus jeunes que leurs fils.

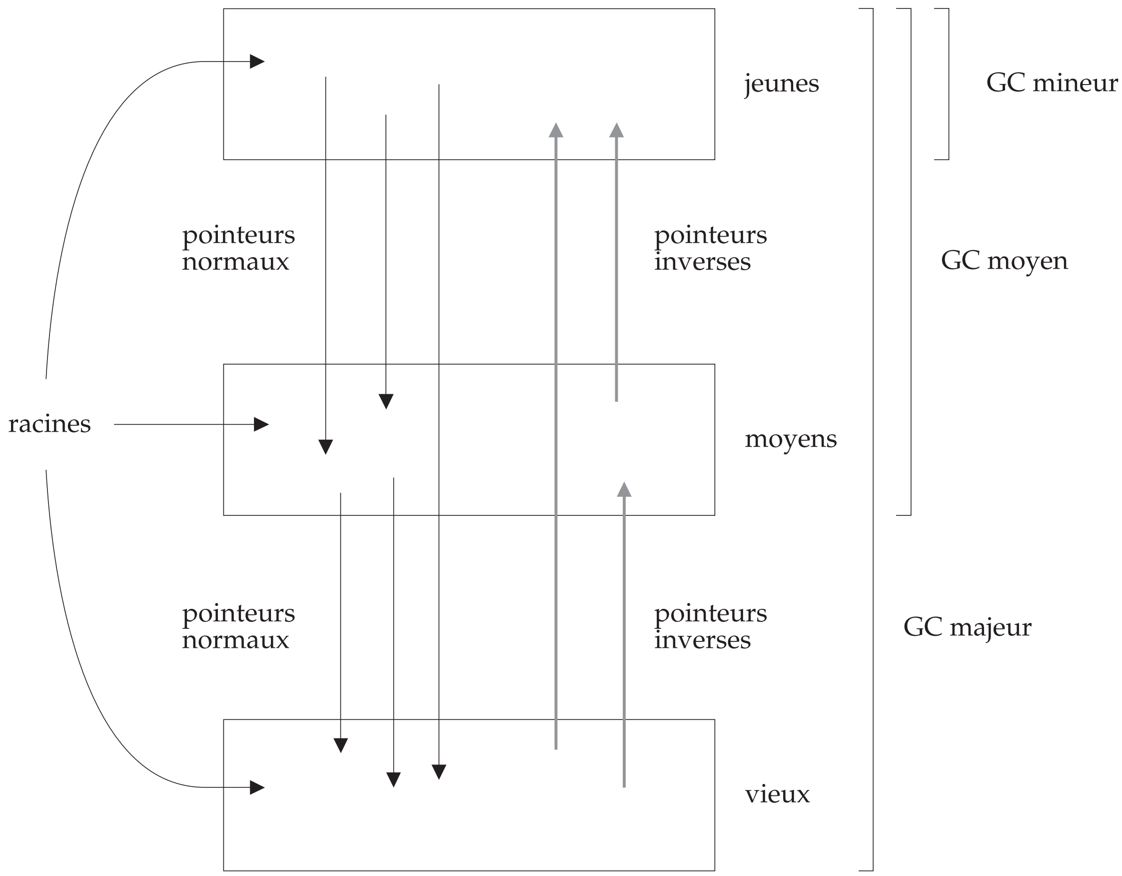


Figure 2.6: Principe du GC à générations (avec 3 générations)

vieux vers les jeunes. Ces pointeurs sont appelés les pointeurs *inverses* et le GC mineur doit les considérer comme des racines.

On peut généraliser ce principe à plus de deux générations. Un cycle de GC de niveau n travaille sur les n générations les plus jeunes, et ses pointeurs inverses sont les pointeurs venant des générations plus vieilles. (figure 2.6) On peut aussi changer la politique de vieillissement en exigeant qu'un objet survive à plusieurs cycles de GC avant de le faire passer dans la génération suivante.

Avantages et inconvénients

Un GC à générations est très efficace, surtout dans les systèmes qui allouent beaucoup d'objets temporaires. Dans ces systèmes, le taux de mortalité des jeunes est très élevé, donc le travail du GC mineur est rentable : la plupart des objets qu'il traite sont inaccessibles, et il récupère donc beaucoup de mémoire.

Un deuxième avantage des générations est le faible temps de latence du GC mineur. En déclenchant le GC mineur assez souvent, on peut s'assurer qu'il a peu de travail à faire (car il n'y a pas beaucoup d'objets jeunes), et donc qu'il n'arrête pas le mutateur trop longtemps.

Le principal inconvénient des générations, c'est qu'il faut que l'affectation dans les objets vieux mette à jour la liste des pointeurs inverses. Cela suppose la coopération du compilateur, et impose un surcoût relativement important sur toutes les opérations d'affectation.

2.3.5 GC parallèles, concurrents, incrémentaux

Sur les machines multi-processeurs, on peut tirer profit de la puissance de calcul disponible de deux façons pas forcément incompatibles (figure 2.7) :

- Un GC *concurrent* exécute le collecteur et le mutateur en même temps sur des processeurs différents.
- Un GC *parallèle* exécute le collecteur sur plusieurs processeurs à la fois. Le mutateur est arrêté pendant le cycle de GC, à moins que le GC ne soit aussi concurrent.

Ces deux techniques sont assez difficiles à mettre au point, mais elles ont un avantage appréciable : le temps de latence diminue fortement, voire disparaît complètement.

Les GC *incrémentaux* sont dérivés des GC concurrents. Au lieu d'exécuter en parallèle le collecteur et le mutateur, on alterne les deux sur un seul processeur. On obtient ainsi les avantages (et les inconvénients) d'un GC concurrent sur une machine séquentielle.

2.3.6 GC conservatifs et racines ambiguës

Il est possible d'ajouter un GC à un langage à désallocation explicite tel que C sans avoir à réécrire le compilateur. En effet, la technique du GC *conservatif* permet d'affaiblir les contraintes imposées par le collecteur sur la structure du graphe mémoire (et donc sur le mutateur) [12, 14, 27, 96]. Le contrat imposé par le collecteur est donc plus facile à satisfaire, à tel point que les programmes “normaux” le remplissent naturellement.

Le contenu d'un objet n'est rien d'autre que des octets. L'interprétation de ces octets varie d'un programme à l'autre et d'un objet à l'autre. Un pointeur est généralement représenté par quelques octets, et un entier est aussi représenté par quelques octets (souvent le même nombre que pour un pointeur). De même les chaînes de caractères et les nombres flottants sont représentés par des octets. Or il n'est pas possible en examinant seulement le contenu des octets de déterminer s'ils représentent un pointeur, un entier, des caractères, etc. Dans un système avec GC non conservatif, le mutateur donne au collecteur le moyen de trouver les pointeurs (le plus souvent en écrivant dans l'objet une description de son contenu).

Dans un système avec GC conservatif, le GC n'a pas besoin de savoir exactement où se trouvent les pointeurs. Il examine tout le contenu de l'objet et il considère comme un pointeur toute suite d'octets qui peut représenter un pointeur vers le tas. Il est donc sûr de trouver tous les vrais pointeurs, avec quelques faux pointeurs en plus. Il applique le même traitement à la pile et aux variables globales pour trouver les racines.

Du point de vue du graphe mémoire, le collecteur travaille donc sur un sur-ensemble des flèches et un sur-ensemble des racines. Il calcule donc un sur-ensemble des objets accessibles (puisqu'il trouve au moins tous les descendants des objets-racines). Il peut donc désallouer les autres, qui représentent un sous-ensemble des objets inaccessibles.

Le GC à *racines ambiguës* est une variante qui ne considère qu'un sur-ensemble des racines. Le contrat que le mutateur doit respecter l'oblige à utiliser une représentation des données qui permet au collecteur de retrouver les pointeurs contenus dans les objets, mais il n'y a pas de contraintes sur le contenu des variables globales et de la pile : le collecteur trouve les racines sans l'aide du mutateur.

Avantages et inconvénients

Les GC conservatifs imposent très peu de contraintes au mutateur, et celui-ci peut alors être compilé par un compilateur quelconque. Les GC à racines ambiguës imposent plus de contraintes sur le mutateur, mais ils sont très utiles dans les langages comme C où le programmeur contrôle complètement le contenu des objets mais pas celui de la pile. En effet, le placement des variables locales dans la pile est fait par le compilateur, qui n'est pas prévu pour donner des informations sur la structure de la pile, tandis que le contenu du tas est géré explicitement par le programmeur, qui peut donc respecter les contraintes imposées par le GC.

Les GC conservatifs sont sujets aux fuites de mémoire. En effet, le GC évite de désallouer les objets qui semblent accessibles. Parmi ces objets, certains sont en fait inaccessibles. Ce problème est rare en pratique, mais il n'y a aucune garantie qu'il ne peut pas devenir grave dans un cas particulier de mutateur malchanceux. Le même problème se pose pour les GC à racines ambiguës, mais il est moins grave car il concerne moins d'objets.

Le plus grave inconvénient des GC conservatifs est qu'ils ne peuvent pas déplacer les objets. En effet, pour déplacer un objet il faut mettre à jour tous les pointeurs vers cet objet. Or le GC ne peut pas distinguer les vrais pointeurs des pseudo-pointeurs, et il n'est pas question de changer les pseudo-pointeurs, qui représentent des données du programme que le GC ne doit pas changer (par exemple un nombre flottant). Cela signifie que les GC conservatifs ne peuvent pas être des GC à copie. De même, un GC à racines ambiguës ne peut pas déplacer les objets-racines.

2.3.7 Allocation

Le travail des primitives d'allocation et de désallocation dépend de la méthode de GC utilisée. Si on utilise un GC à balayage ou à comptes de références, la mémoire sera fragmentée. Le GM doit alors gérer une *liste libre* : une structure de données qui permet à la fonction d'allocation de trouver une zone libre de taille suffisante pour allouer l'objet demandé par le mutateur. La fonction de désallocation insère les blocs libérés dans cette liste. La liste libre est généralement une liste chaînée (d'où son nom), mais on peut aussi utiliser des structures d'arbres pour faciliter la recherche. Il existe de nombreuses stratégies d'allocation qui visent à minimiser le temps de recherche et la fragmentation. On en trouve une bonne description dans [87, chapitre 5].

Si on utilise un GC à copie, la fonction d'allocation est beaucoup plus simple (et plus rapide). En effet, la mémoire libre est composée d'une zone contiguë. Pour allouer un objet il suffit de couper un bloc de la bonne taille au début de cette zone libre. La fonction d'allocation est tellement simple qu'on peut la *déplier*, c'est-à-dire remplacer l'appel par le corps de la fonction dans le code du mutateur. Cette technique permet d'améliorer encore les performances.

Si on a un GC à générations on utilise souvent une zone mineure de petite taille pour obtenir un temps de latence faible. Dans ce cas, les objets plus gros que la zone mineure ne peuvent pas être alloués comme les objets normaux dans la zone mineure. Il faut donc un cas spécial de la fonction d'allocation pour les allouer directement dans le tas majeur. Notons que ce cas spécial n'empêche pas de déplier la fonction d'allocation si la taille de l'objet à allouer est connue au moment de la compilation, ce qui est le cas pour la plupart des allocations en ML.

2.4 Choix d'une stratégie de GC

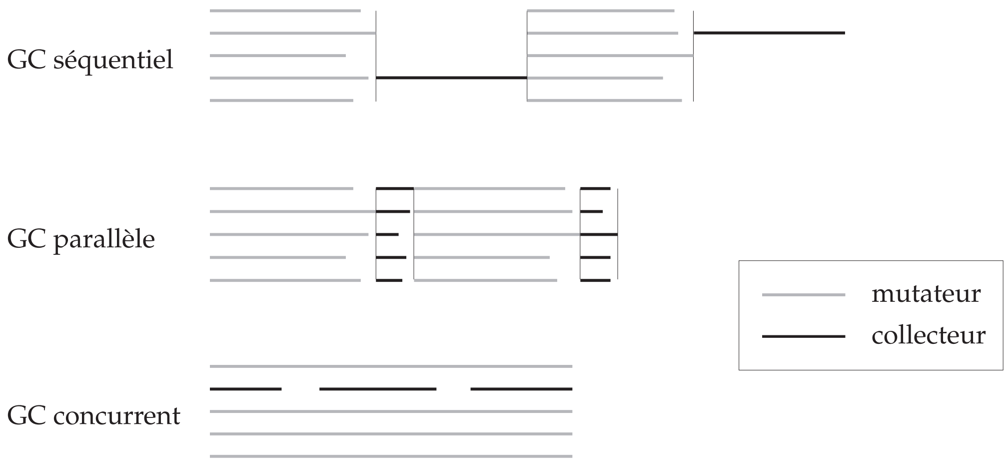
On peut mélanger les techniques de base en n'importe quelle combinaison (ou presque). Le choix des ingrédients et des proportions dépend surtout des contraintes extérieures énumérées en section 2.2.

Caractéristiques du langage

Les critères les plus importants pour le choix d'une technique de GC sont les taux d'allocation et de mutation du langage. Un fort taux d'allocation s'accompagne généralement d'un fort taux de mortalité, ce qui avantage les GC à copie car il y a alors beaucoup d'objets morts dans le tas. Dans ces conditions, un GC à copie utilise peu de temps de calcul (pour copier les objets vivants peu nombreux) et récupère beaucoup de mémoire (occupée par les nombreux objets morts). L'allocation simplifiée apportée par un GC à copie est aussi un avantage important puisqu'on fait beaucoup d'allocations.

Un fort taux d'allocation est souvent associé à une utilisation intensive d'objets temporaires à courte durée de vie. Cette caractéristique avantage les GC à générations, car le GC mineur, peu coûteux, désalloue beaucoup d'objets. Le GC à copie est particulièrement adapté pour un GC mineur, car il est rapide dans le cas où il y a peu d'objets vivants (ce qui est le cas dans la zone mineure). De plus le vieillissement se fait directement par la copie si on place les jeunes et les vieux dans deux zones distinctes de la mémoire. Enfin, l'utilisation d'un GC mineur à copie donne une allocation simplifiée dans la zone mineure, ce qui réduit le coût de la fonction d'allocation.

Un faible taux de mutation avantage aussi les GC à générations et les GC concurrents ou incrémentaux puisqu'ils imposent un surcoût à l'affectation, qui est rarement utilisée.



Chaque ligne horizontale représente le travail effectué par un processeur au cours du temps.

Figure 2.7: Comparaison des GC séquentiel, parallèle et concurrent sur une machine à 5 processeurs. Le GC séquentiel laisse 4 processeurs inexploités pendant son temps de latence.

Parallélisme

Sur une machine parallèle, il faudra choisir un GC parallèle ou concurrent, sous peine de diminuer les performances du programme de façon inacceptable. On le voit sur la figure 2.7, où la même quantité de travail est représentée dans les trois cas. Le programme prend beaucoup plus de temps dans le cas du GC séquentiel.

Portabilité

Certains GC à copie incrémentaux ou concurrents [6] utilisent des fonctions spéciales du système d'exploitation pour intercepter les lectures et écritures du mutateur dans les objets en cours de copie. Cette technique évite au mutateur de tester à chaque accès si l'objet accédé est en cours de copie ; elle est donc potentiellement très efficace.

Ces algorithmes posent cependant des problèmes de portabilité, car les fonctions qu'ils utilisent ne sont pas toujours disponibles, leur interface change d'un système à l'autre, et leurs performances sont très variables [8]. Il est donc préférable de les réserver aux cas où la portabilité du système n'est pas un critère important.

Un autre exemple de technique non portable est la spécialisation de la machine par l'ajout de quelques instructions destinées à faciliter l'implémentation du GC [74, 90]. Cette technique n'est plus guère utilisée car la conception et la fabrication des microprocesseurs est de plus en plus coûteuse. Les machines spécialisées ne sont donc plus rentables.

Temps de latence et performances

Les programmes temps-réel ou interactifs exigent des temps de latence courts. Pour obtenir ce résultat, il s'avère indispensable d'utiliser un GC concurrent ou incrémental. Un système à générations peut aussi être très utile dans ce cas.

Dans tous les cas, on veut aussi que le temps de calcul du GC soit assez faible par rapport à celui du mutateur. Ce résultat est d'autant plus difficile à obtenir que le taux d'allocation est grand : si le programme consomme beaucoup de mémoire, il faut la recycler plus vite, et le GC a donc plus de travail à faire. Heureusement, un taux d'allocation élevé avantage les GC à génération et à copie, qui sont très efficaces.

Influence du compilateur

Si on désire ajouter un GC à un compilateur préexistant, il faudra choisir un GC conservatif ou à racines ambiguës.

Ces techniques s'avèrent aussi très utiles pour utiliser un compilateur d'un langage de bas niveau comme C en tant que générateur de code pour un autre langage. En effet, on peut écrire un compilateur pour un langage à désallocation implicite (par exemple ML ou Scheme) qui génère du code C. Le code C généré par le compilateur ML peut satisfaire les contraintes imposées par le GC sur le contenu des objets, mais il ne contrôle pas le contenu de la pile, qui est gérée par le compilateur C. On peut donc utiliser un GC à racines ambiguës. Cette technique permet d'obtenir des compilateurs ML relativement efficaces et facilement portables, puisque le code généré ne dépend pas de la machine (il suffit qu'elle dispose d'un compilateur C).

2.5 Les stratégies utilisées dans cette thèse

Les stratégies retenues pour les GM décrits dans cette thèse sont pour la plupart dictées par l'utilisateur de ces GM : le système Caml Light. Ce système est décrit dans le chapitre suivant (section 3.3.1). Il impose naturellement les choix suivants :

- Le compilateur contrôle le contenu des objets et de la pile, donc on n'a pas besoin d'un GC conservatif ou à racines ambiguës.
- Caml Light est un dialecte de ML, il a donc un faible taux de mutation et un fort taux d'allocation ; de plus, le taux de mortalité des objets jeunes est très grand. Ces caractéristiques avantagent largement un GC à générations.
- C'est le GC mineur qui désalloue la plupart des objets, il faut donc qu'il soit le plus efficace possible. De plus, le nombre d'objets jeunes vivants au moment du GC mineur est faible (toujours à cause du taux de mortalité des objets jeunes). On utilisera donc un GC mineur à copie.
- Le système Caml Light est hautement portable. Nous évitons donc d'utiliser des fonctions spéciales du système d'exploitation ou toute autre technique qui rendrait le portage difficile.

Nous avons moins de contraintes sur le GC majeur que sur le GC mineur, et nous avons donc exploré plusieurs combinaisons :

- un GC hybride à copie et à balayage, décrit dans le chapitre 3,
- un GC concurrent pour machines parallèles qui a permis de produire une version parallèle de Caml Light, décrit dans le chapitre 7,
- un GC incrémental dérivé du GC concurrent qui permet d'écrire des programmes interactifs en Caml Light, décrit à la fin du chapitre 7.

Chapitre 3

GC hybride avec une génération

Ce chapitre présente le GC utilisé dans les versions 0.3 et 0.4 de Caml Light. Il s'agit d'un GC à générations ; le GC mineur est un GC à copie et le GC majeur est un hybride de copie et de balayage qui utilise l'algorithme de Lang et Dupont. La section 3.1 développe l'algorithme hybride de Lang et Dupont, la section 3.2 décrit le GC mineur et son interaction avec le GC majeur, la section 3.3 explique les problèmes rencontrés et les solutions apportées lors de l'implémentation de ce système, et la section 3.4 donne des mesures de performances et tire les conclusions de cette expérience.

3.1 L'algorithme de Lang et Dupont

Dans [62], Lang et Dupont présentent une synthèse des algorithmes de copie et de balayage. En faisant abstraction des détails de la représentation du graphe mémoire, de la gestion du tas, et de la méthode de parcours du graphe, ils obtiennent une présentation uniforme, que l'on appellera l'*algorithme abstrait*, que l'on peut implémenter aussi bien par un GC à copie que par un GC à balayage. Ces deux techniques, qui apparaissaient comme complètement différentes, deviennent alors simplement des implémentations de l'algorithme abstrait. L'algorithme abstrait est a priori compatible avec toutes les variantes données au chapitre 2 : on peut envisager des versions conservatives, concurrentes, parallèles, etc.

L'algorithme abstrait a un autre avantage : il peut s'implémenter par un GC hybride, qui copie certains objets et balaye les autres, la proportion pouvant varier d'un cycle à l'autre. Le travail d'abstraction à partir des deux algorithmes de base a donc permis l'invention d'un nouvel algorithme, dont les propriétés sont très intéressantes.

Algorithme abstrait

On distingue trois ensembles d'objets dans la mémoire : les objets *parcourus*, les objets *marqués*, et les objets *non-marqués*. Ces trois ensembles généralisent les trois couleurs utilisées en section 2.3.2. Le GC fait passer les objets d'un ensemble à l'autre pour trier les objets accessibles et inaccessibles. (figure 3.1)

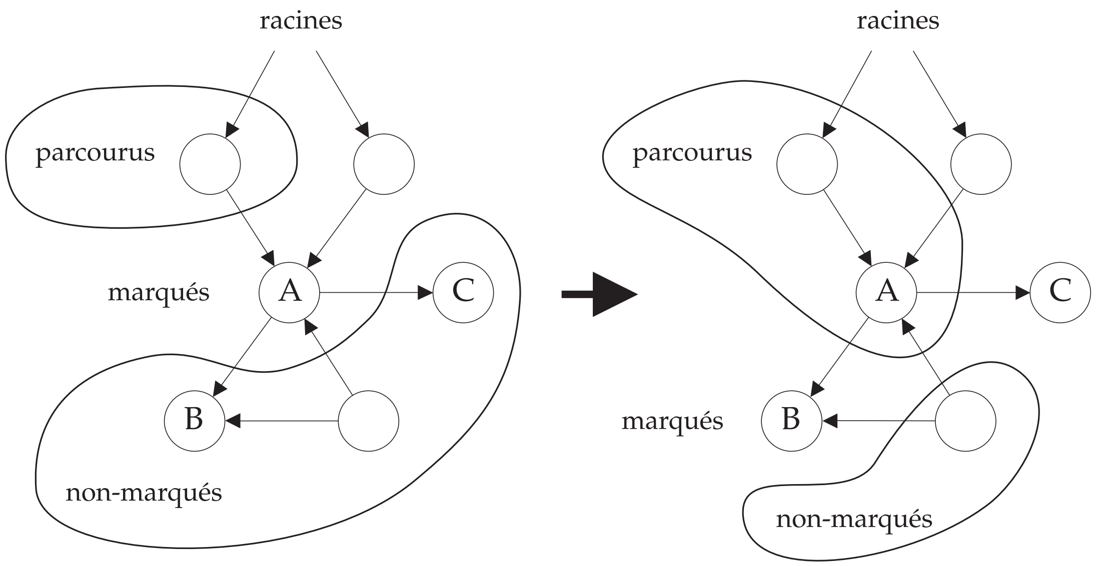


Figure 3.1: L'algorithme de Lang et Dupont : parcours de A et marquage de ses fils B et C

Au début du cycle de GC, tous les objets de la mémoire sont dans l'ensemble des non-marqués. Le cycle commence par une phase de parcours du graphe mémoire qui identifie les objets vivants : on fait passer les racines dans l'ensemble des marqués, puis on répète l'opération suivante : prendre un objet marqué, marquer tous ses fils non-marqués, et le mettre dans l'ensemble des objets parcourus, jusqu'à ce qu'il n'y ait plus d'objet marqué. Alors tous les objets accessibles (et eux seuls) sont parcourus, et la phase de parcours est finie. Remarquons qu'au cours de la phase de parcours, 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 de GC.

La deuxième phase est la phase de désallocation : on désalloue tous les objets non-marqués puis on remet tous les objets vivants dans l'ensemble des non-marqués pour préparer le cycle suivant.

Spécialisations de l'algorithme abstrait

Le GC à copie peut être considéré comme un cas particulier de l'algorithme de Lang et Dupont : 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 parcours, et l'ensemble des objets marqués est la portion de l'espace d'arrivée comprise entre le pointeur de copie et le pointeur de parcours (figure 2.5). Les objets non-marqués deviennent marqués lorsqu'ils sont copiés et les objets marqués deviennent parcourus avec l'avance du pointeur de parcours. La phase de désallocation consiste simplement à échanger espace de départ et espace d'arrivée : les objets morts sont laissés dans l'espace de départ et ils sont donc désalloués en masse.

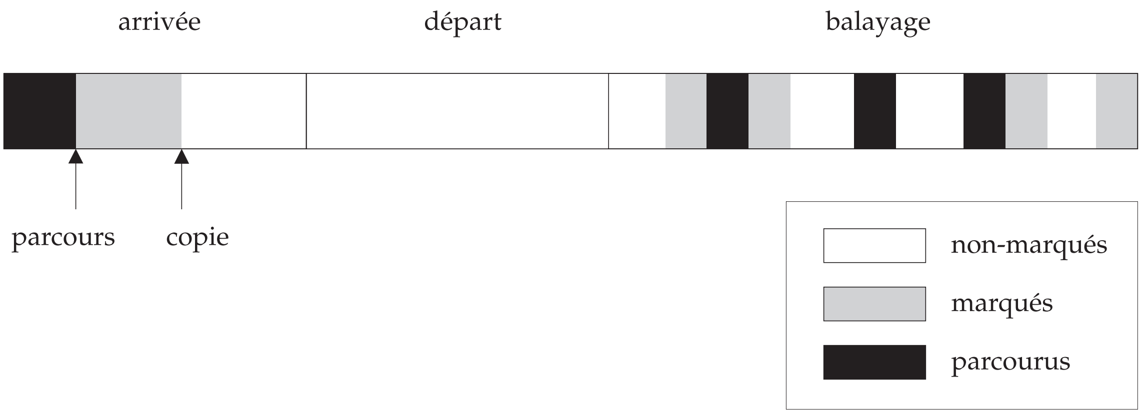


Figure 3.2: Principe de l'algorithme hybride

Le GC à balayage est lui aussi un cas particulier de l'algorithme de Lang et Dupont. Avec un marquage à trois couleurs, les objets non-marqués sont blancs, les objets marqués sont gris et les objets parcourus sont noirs. Avec un marquage à deux couleurs, les objets non-marqués sont blancs, les objets marqués sont les noirs dont le collecteur n'a pas fini de parcourir les fils, et les objets parcourus sont les noirs dont il a fini de parcourir les fils. La phase de désallocation coïncide avec le balayage : les objets blancs (non-marqués) sont désalloués et les objets noirs (parcourus) sont blanchis (replacés dans l'ensemble des non-marqués).

Algorithme hybride

L'algorithme de Lang et Dupont peut aussi s'implémenter par un GC dit *hybride*, c'est-à-dire qui mélange copie et balayage. On partitionne l'ensemble des objets en trois sous-ensembles : l'espace de départ, l'espace d'arrivée et l'espace de balayage. L'espace d'arrivée doit être vide au début du GC, et il est partitionné en trois à l'aide de deux pointeurs (parcours et copie), comme l'espace d'arrivée d'un GC à copie. Chaque objet a une couleur : noir, gris ou blanc, comme dans un GC à balayage.

Les trois ensembles de l'algorithme abstrait sont représentés comme suit :

- L'ensemble des objets non-marqués correspond aux objets de l'espace de départ et aux objets blancs de l'espace de balayage.
- Les objets marqués sont les objets de l'espace d'arrivée situés entre les pointeurs de parcours et de copie et les objets gris de l'espace de balayage.
- Les objets parcourus sont les objets de l'espace d'arrivée situés avant le pointeur de parcours et les objets noirs de l'espace de balayage.

L'opération de parcours d'un objet A consiste à marquer tous ses fils, puis :

- si A est dans l'espace d'arrivée, à avancer le pointeur de parcours (A doit alors être le premier objet après le pointeur de parcours),

- si A est dans l'espace de balayage, à changer sa couleur en noir.

L'opération de marquage d'un objet A consiste, s'il n'est pas déjà marqué :

- s'il est dans l'espace de départ, à le copier dans l'espace d'arrivée (en déplaçant le pointeur de copie, comme dans un GC à copie),
- s'il est dans l'espace de balayage, à changer sa couleur en gris.

La phase de parcours commence par marquer les objets-racines, puis elle répète "trouver un objet marqué et le parcourir" jusqu'à ce qu'il n'y ait plus que des objets parcourus (ce sont les objets accessibles) ou non-marqués (ce sont les objets inaccessibles).

Comme dans un GC à copie, à chaque fois que le GC trouve un pointeur p vers un objet A de l'espace de départ qui a déjà été copié, il doit remplacer p par un pointeur vers la copie A' de A. Il trouve cette copie en suivant le pointeur de renvoi.

La phase de désallocation consiste à désallouer d'un coup tout l'espace de départ (puisqu'il ne contient plus d'objet vivant), et à balayer l'espace de balayage pour désallouer les objets blancs et blanchir les objets noirs pour préparer le cycle suivant.

Il faut aussi que la copie colorie en blanc les objets copiés pour que tous les objets soient blancs à la fin du cycle. Le cycle suivant peut alors choisir de nouveaux espaces de départ, d'arrivée et de balayage, ce qui permet de faire changer dynamiquement les parties du tas traitées en copie et en balayage. On appelle la *proportion de balayage* le pourcentage du tas occupé par l'espace de balayage. Cette proportion peut changer d'un cycle de GC à l'autre.

Avantages et inconvénients

L'algorithme hybride combine les avantages et les inconvénients des GC à copie avec ceux des GC à balayage. Parmi les avantages, on a un compactage partiel de la mémoire, puisque les objets de l'espace de départ sont copiés dans l'espace d'arrivée ; celui-ci n'est donc pas fragmenté. Si on utilise une proportion de balayage de 0%, alors le GC hybride est un GC à copie pur. Il est donc très rapide, mais il réserve la moitié de la mémoire (pour l'espace d'arrivée). Si on veut économiser la mémoire, on peut augmenter la proportion de balayage, au prix d'un GC plus lent et d'une fragmentation partielle du tas. On peut cependant maintenir cette fragmentation à un niveau acceptable en déplaçant l'espace de départ pour compacter une zone différente à chaque cycle (figure 3.3).

Les inconvénients du GC hybride sont aussi une combinaison de ceux des GC à copie et 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 assez compliquée.

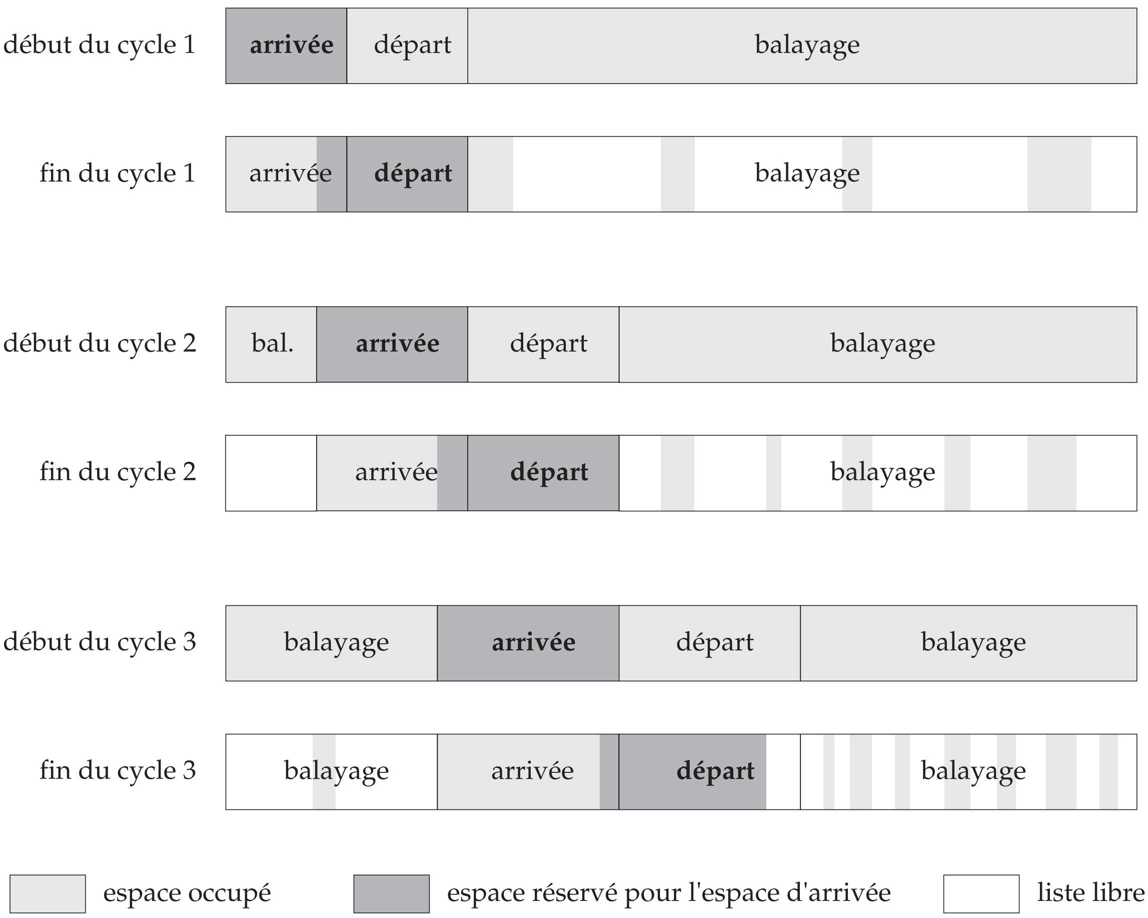


Figure 3.3: Compactage incrémental avec l'algorithme hybride

3.2 Ajout d'une génération

L'ajout d'une génération est en principe très simple : il suffit de choisir une technique pour le GC mineur et de combiner ce GC mineur avec un GC majeur hybride. Le choix du GC mineur est primordial pour le langage ML à cause de son taux d'allocation élevé. En effet, la plupart des objets ont une durée de vie très courte, donc le GC mineur désalloue la plupart des objets. C'est donc lui qui va faire la plus grande partie du travail de GC.

Choix du GC mineur

Il y a plusieurs raisons d'utiliser un GC mineur à copie :

- Le GC à copie est très économique dans le cas où il y a peu d'objets vivants.
- La copie permet de représenter la zone mineure par une zone contiguë de la mémoire. Le vieillissement consiste alors à copier l'objet hors de cette zone, et il

est facile de tester l'appartenance d'un objet à la zone mineure.

- L'allocation est peu coûteuse et facile à implémenter.

Cette dernière raison est la plus importante. En effet, le mutateur alloue beaucoup d'objets. Il fait donc souvent appel à la fonction d'allocation et il est important qu'elle soit rapide. De plus, comme son code ne représente que quelques lignes, on peut la déplier dans le code du mutateur dans le cas (très fréquent) où la taille de l'objet à allouer est connue à la compilation. On évite ainsi le coût d'un appel de fonction à chaque allocation.

Interfaces entre le mutateur et les collecteurs

Avec un GC mineur à copie, on peut avoir une vision modulaire du système de gestion de mémoire :

- Le GC majeur et les primitives d'allocation dans le tas majeur fournissent un service de gestion de mémoire (le GM majeur) qui est utilisé par le reste du système (GC mineur + mutateur).
- Le GC mineur utilise ce service pour réallouer dans le tas majeur les objets qu'il copie, et avec les primitives d'allocation dans le tas mineur il fournit à son tour un service de gestion de mémoire (le GM mineur) qui est utilisé par le mutateur.
- Le mutateur utilise les services du GM mineur pour allouer la plupart des objets et ceux du GM majeur pour les objets qui sont trop gros pour être alloués dans le tas mineur.

Le point de vue du GC majeur est donc que l'ensemble GC mineur + mutateur est un mutateur auquel il fournit des services de gestion de mémoire. Le point de vue du mutateur est que l'ensemble GM mineur + GM majeur est un GM dont il utilise les services.

3.3 Détail de l'implémentation

Cette section présente une vue détaillée du GC hybride avec une génération, expose les problèmes rencontrés et les solutions retenues.

3.3.1 Le système Caml Light

Nous allons décrire brièvement les traits du système Caml Light qui ont une influence sur l'implémentation du GM. On trouve une description plus complète de Caml Light dans [95, 68, 66, 67].

Le langage Caml Light

Caml Light est un langage de la famille ML. Il s'agit d'un langage de très haut niveau qui est particulièrement adapté à la manipulation d'objets symboliques comme les listes, les graphes, les arbres, les termes, etc. Dans ce langage, la gestion de la mémoire est complètement à la charge du système et le programmeur n'a pas à s'en soucier. Même les fonctions d'allocation sont appelées implicitement, par un certain nombre de constructions du langage.

Ces caractéristiques, la technique de compilation utilisée et le style normal de programmation en Caml Light impliquent un fort taux d'allocation. De plus, à cause du système de types de ML, les opérations d'affectation sont restreintes à une classe d'objets dits *mutables*. L'utilisation des objets mutables est soumise à des contraintes particulières qui découragent leur emploi. Le programmeur est donc encouragé à allouer un nouvel objet pour chaque résultat intermédiaire plutôt que d'utiliser des objets mutables. Ces restrictions sur les objets mutables donnent au langage un faible taux de mutation.

L'implémentation de Caml Light

Les versions 0.3 et 0.4 de Caml Light sont implémentées par un compilateur qui produit du code pour une machine virtuelle à pile : le *byte-code*. Ce code est ensuite interprété par un programme écrit en C. Cette technique permet d'obtenir un système extrêmement facile à porter sur une nouvelle machine. En effet, l'interpréteur de byte-code est écrit en C hautement portable, et le compilateur est complètement indépendant de la machine.

L'interpréteur de byte-code s'interface avec le GM de façon naturelle, puisque celui-ci est aussi écrit en C. La machine virtuelle de Caml Light utilise deux piles pour stocker les arguments des fonctions et les variables locales. L'interpréteur de byte-code fournit au GM une fonction qui lui permet de trouver les pointeurs contenus dans les piles (ces piles sont distinctes de la pile utilisée par le compilateur C pour stocker les variables locales de l'interpréteur). Le compilateur Caml Light place toutes les variables globales du code Caml dans un objet spécial du tas, dont l'adresse est stockée dans une variable globale C commune au GM et à l'interpréteur : `global_data`. Tous les accès aux variables globales de Caml se font donc par indirection à partir de `global_data`. En conséquence, le GM a affaire à un mutateur qui n'a qu'une variable globale : les racines sont `global_data` et le contenu des piles.

Interface avec le code C

L'interpréteur de byte-code étant écrit en C, il s'interface facilement avec des fonctions C, et le compilateur Caml Light permet au code Caml d'appeler des fonctions écrites en C. Cela permet d'utiliser des bibliothèques de fonctions C dans un programme Caml Light : X windows, entiers en précision arbitraire, BDD (*binary decision diagrams*), expressions régulières, etc. Cela signifie aussi que le GM doit cohabiter avec la bibliothèque standard de C et son GM, qui est un système à désallocation explicite basé

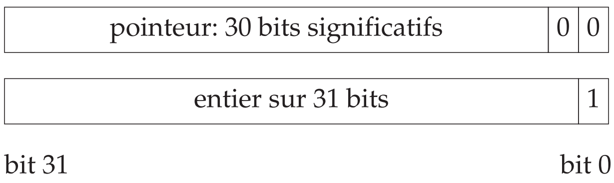


Figure 3.4: Codage des valeurs

sur `malloc` et `free`.

3.3.2 Organisation de la mémoire

Le GC de Caml Light 0.4 n'est pas un GC conservatif. Il impose donc au contenu de la mémoire une structure qui lui permet de trouver les pointeurs.

Valeurs et objets

Caml Light 0.4 est fait pour fonctionner sur machines 32 bits, c'est-à-dire sur les machines où les pointeurs et les entiers occupent 4 octets. L'accès à la mémoire se fait par *mots* de 32 bits. Les objets de Caml Light sont tous constitués d'un nombre entier de mots, et ils sont toujours *alignés* : leur adresse est un multiple de 4 octets.

On appelle *valeur* le contenu d'un mot de 32 bits. Caml Light travaille sur deux classes de valeurs : les pointeurs et les entiers. Le GC les distingue par leur bit de poids faible : les pointeurs sont pairs (puisqu'ils sont multiples de 4), et les entiers sont les valeurs impaires. Pour pouvoir travailler avec des entiers pairs, l'interpréteur de byte-code stocke un entier de 31 bits dans les bits de poids fort d'une valeur dont le bit de poids faible est 1. Un entier n est donc codé par la valeur $v = 2n + 1$ (figure 3.4).

Chaque objet est constitué d'un en-tête qui occupe un mot et d'un *corps* qui occupe un nombre entier non nul de mots. Un pointeur vers un objet est l'adresse du premier octet du corps. Les pointeurs vers l'en-tête ou les autres octets du corps sont considérés comme des pointeurs infixes. Ils ne doivent donc apparaître ni dans le graphe mémoire ni parmi les racines.
L'en-tête occupe un mot. Il contient :

- Le *type d'objet* (sur 8 bits), qui code le constructeur de la valeur ML.
- La couleur (sur 2 bits), qui sert au GC.
- La taille (sur 22 bits), qui donne le nombre de mots du corps.

Selon le type d'objet, le contenu du corps peut être :

- Des champs, qui sont des valeurs (pointeurs ou entiers). Chaque champ occupe un mot. On dit alors que l'objet est *structuré*.

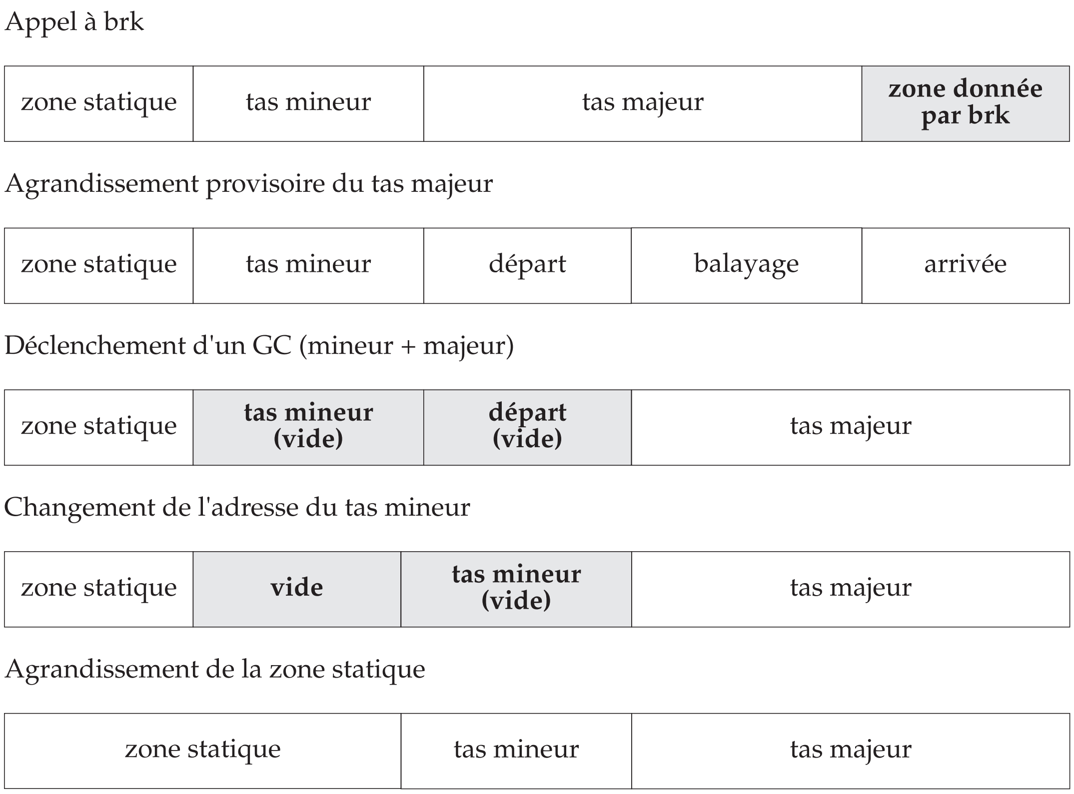


Figure 3.5: Décalage du tas

- Des données brutes, qui ne seront pas examinées par le GC (par exemple un nombre flottant ou une chaîne de caractères). On dit alors que l'objet est *brut*. Un objet brut ne doit pas contenir de pointeurs car le GC ne pourrait pas les mettre à jour quand il déplace les objets correspondants.

Le collecteur distingue ces deux cas en comparant le type d'objet avec la constante `NO_SCAN_TAG`. Les types d'objets supérieurs ou égaux à cette constante correspondent aux objets bruts ; les types d'objets inférieurs désignent les objets structurés.

Placement des zones du tas

La mémoire est constituée de trois zones : la *zone statique*, le tas mineur et le tas majeur. La réunion des tas mineur et majeur constitue le tas.

La zone statique est gérée par désallocation explicite, le GC n'a donc pas besoin de l'explorer. Elle contient le byte-code, les piles de l'interpréteur de byte-code, les tables utilisées par le GC, etc. On peut placer dans les objets du tas des pointeurs vers la zone statique : le GC les traite comme les entiers, c'est-à-dire qu'il les ignore.

Le placement des zones est indiqué en haut de la figure 3.5. Le tas mineur est de taille fixe. Le GM peut agrandir le tas majeur et la zone statique pour satisfaire les requêtes d'allocation. L'agrandissement du tas majeur se fait simplement avec l'appel système `brk`, qui agrandit la mémoire disponible du processus. L'agrandissement de la zone statique est plus compliqué, car la mémoire donnée par `brk` se trouve à la fin du tas majeur. Il faut donc décaler celui-ci et le tas mineur pour arriver à agrandir la zone statique. Pour décaler le tas mineur, il suffit de le vider (par un GC mineur) et de changer son adresse ; pour décaler le tas majeur, il faut effectuer un GC majeur avec des espaces de départ et d'arrivée bien choisis (figure 3.5).

Interface entre le GM et le mutateur

Le mutateur est constitué de l'interpréteur de byte-code et du programme qu'il interprète. Son interface avec le GM comporte :

- la constante `NO_SCAN_TAG`,
- les fonctions `alloc_small` et `alloc`, qui permettent au mutateur d'allouer des objets respectivement dans le tas mineur et dans le tas (majeur ou mineur selon la taille de l'objet),
- la macro `MODIFY`, fournie par le GM au mutateur ; celui-ci doit l'utiliser à la place de l'opération d'affectation de C pour toutes les affectations dans les objets structurés (son rôle sera expliqué dans la section 3.3.5),
- la fonction `trace_roots` que le mutateur fournit au GM. Elle doit énumérer les racines.

De plus, le mutateur doit respecter les contraintes suivantes :

- Il ne modifie pas la taille ou la couleur d'un objet.
- Il ne stocke pas de pointeur infixe parmi les racines ou dans les objets structurés. Il peut utiliser des pointeurs infixes temporairement, par exemple un pointeur dans une chaîne de caractères pour la recopier, mais ces pointeurs sont invalidés (car les objets sont déplacés) par les fonctions d'allocation (car elles peuvent appeler le GC).
- Il ne stocke pas de pointeurs vers le tas dans les objets bruts.

3.3.3 Choix des espaces de départ et d'arrivée

Le GC majeur doit choisir la taille et la position dans le tas des espaces de départ et d'arrivée. Il dispose donc d'un degré de liberté que n'ont pas les algorithmes à copie ou à balayage.

Il doit choisir l'espace d'arrivée à la fin du cycle pour le cycle suivant, car cet espace est réservé au GC : il ne faut pas allouer dedans. La taille de l'espace de départ est la même que celle de l'espace d'arrivée. En effet, si on prend un espace de départ plus grand et si tous ses objets sont vivants, l'espace d'arrivée va déborder et le GC ne pourra pas se terminer ; si on prend un espace de départ plus petit, on a gaspillé la mémoire libre en la réservant pour le GC alors qu'il n'en a pas besoin.

Il faut aussi choisir l'emplacement de l'espace d'arrivée, qui doit être une zone de mémoire libre contiguë, et celui de l'espace de départ. L'implémentation de Caml Light 0.4 utilise l'algorithme suivant, qui est largement arbitraire, mais qui donne de bons résultats :

- À la fin du cycle de GC, il mesure le taux d'occupation du tas, agrandit le tas si nécessaire pour maintenir le taux d'occupation à un niveau raisonnable, puis il détermine la proportion de balayage idéale en fonction du nouveau taux d'occupation.

- Il recolle l'ancien espace de départ avec les zones libres adjacentes, puis il réserve une partie de cette zone pour l'espace d'arrivée du cycle suivant. La taille de cette partie est déterminée par la proportion de balayage idéale calculée précédemment.

- Au début du cycle suivant, il place l'espace de départ juste à côté de l'espace d'arrivée. Il y a deux raisons à ce placement : il permet de recoller la partie libre de l'espace d'arrivée avec l'espace de départ et il permet de compacter incrémentalement tout le tas, comme sur la figure 3.3.

Cette méthode assure que le GC hybride fonctionne exactement comme un GC à copie quand la proportion de balayage est nulle : il alterne les espaces de départ et d'arrivée entre les deux moitiés du tas.

Il se pose un problème pour déterminer la fin exacte de l'espace de départ. En effet, celui-ci doit se finir entre deux objets : on ne peut pas copier une partie d'un objet et marquer le reste. Il faut donc trouver l'objet qui est à cheval sur la frontière "naturelle" de l'espace de départ (celle qui lui donne la même taille que l'espace d'arrivée), et placer la fin de l'espace de départ juste avant cet objet. On peut obtenir ce résultat en balayant l'espace de départ, mais cela annulerait l'avantage de la copie (qui est efficace justement parce qu'elle évite de balayer le tas). La solution retenue est que la fonction de marquage teste chaque objet (au moment de le marquer) pour déterminer s'il s'agit de l'objet à cheval sur la frontière. Lorsque cet objet est trouvé, la frontière est ajustée pour être juste avant cet objet. Au prix d'un test supplémentaire sur chaque objet marqué, on préserve donc une caractéristique intéressante du GC hybride : si la proportion de balayage est nulle, le GC est aussi efficace qu'un GC à copie simple.

3.3.4 Interaction avec `malloc`

Pour coexister avec les fonctions des bibliothèques C utilisées par les programmes ML, le GM doit fournir des fonctions `malloc` et `free`. Or la fonction `malloc` standard appelle `brk` et fait l'hypothèse qu'elle est la seule à appeler `brk`. Le GM fait la même hypothèse, donc il n'est pas compatible avec la fonction `malloc` standard.

Il a donc fallu implémenter un nouveau `malloc`, qui alloue dans la zone statique, et qui décale le tas au besoin. Cette solution n'est pas entièrement satisfaisante, puisqu'elle exige de remplacer les fonctions `malloc` et `free` de la bibliothèque standard, et cette

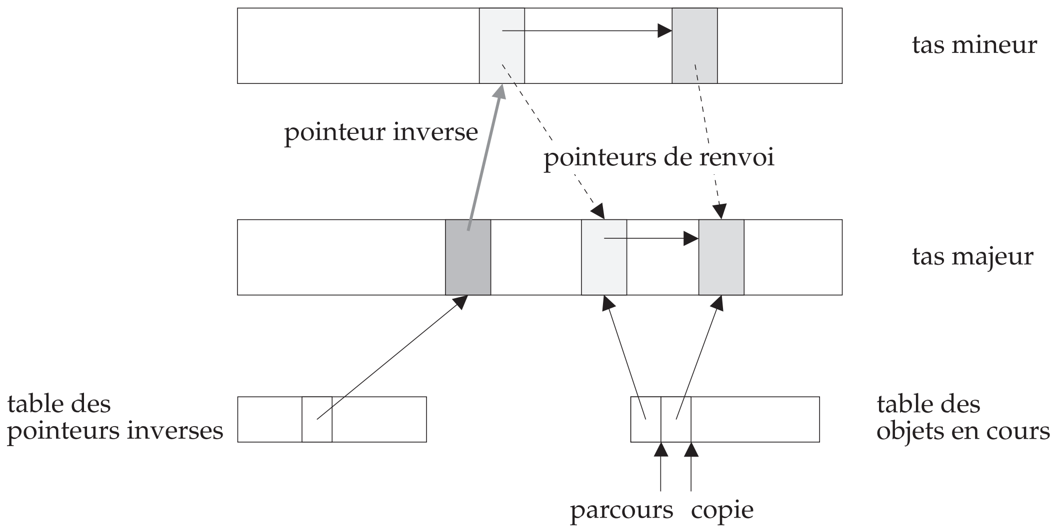


Figure 3.6: Les tables du GC mineur pointent vers des objets du tas majeur.

opération de remplacement est difficile dans l'environnement de programmation en C disponible sous Unix. Son fonctionnement n'est donc pas garanti, ce qui nuisait à la portabilité de Caml Light (versions 0.3 et 0.4).

3.3.5 Interaction entre GC mineur et GC majeur

Comme expliqué dans la section 2.3.4, le GC mineur a besoin de connaître les pointeurs inverses. Il utilise pour cela une table allouée statiquement, dans laquelle `MODIFY` stocke des pointeurs vers les pointeurs inverses (ce sont donc des pointeurs infixes[1]). Le GC mineur utilise les pointeurs indiqués par cette table comme racines supplémentaires. À la fin du cycle de GC mineur, elle est vidée car tous les objets jeunes vivants ont été copiés dans le tas majeur et il n'y a donc plus de pointeurs inverses. Si cette table déborde (parce qu'il y a eu trop d'appels à `MODIFY` entre deux GC mineurs), on déclenche un GC mineur qui videra la table.

Le GC mineur utilise aussi une table des objets en cours. En effet, ceux-ci ne sont pas adjacents dans le tas majeur, puisque l'espace libre du tas majeur est fragmenté. Le GC mineur utilise donc un pointeur de copie et un pointeur de parcours qui sont des indices dans cette table plutôt que des pointeurs vers les objets eux-mêmes.

Le GC majeur se déclenche lorsqu'une requête d'allocation dans le tas majeur ne peut pas être satisfaite (car il n'y a plus de bloc de taille suffisante dans la liste libre). La plupart du temps, cela arrive au cours du GC mineur, puisque celui-ci est le plus gros client du GM majeur. Ce déclenchement du GC majeur pendant le GC mineur pose le problème de la mise à jour des deux tables du GC mineur. En effet, le GC majeur peut déplacer ou même désallouer des objets pointés par ces tables (figure 3.6).

[1] La notion de pointeur infixe est définie en page 17

programme	rendement	latence		temps de calcul
		moyenne	maximale	
KB	97 %	3,0 ms	8,8 ms	8,2 %
compilateur	80 %	2,4 ms	12 ms	3,0 %
CoQ	93 %	1,2 ms	21 ms	2,3 %
SIMPLE	97 %	0,54 ms	4,9 ms	1,6 %

Figure 3.7: Performances du GC mineur

La solution retenue est que le GC majeur met à jour les tables du GC mineur : il utilise la table des objets en cours comme un ensemble de racines, ce qui lui permet de la mettre à jour s'il déplace un objet en cours, et de s'assurer qu'il ne va pas désallouer un objet que le GC mineur veut accéder par la suite. Cette technique ne peut pas s'appliquer à la table des pointeurs inverses, puisque cette table contient des pointeurs infixes, mais le GC majeur peut la reconstituer, puisqu'il parcourt tout le graphe mémoire : il examine tous les pointeurs, donc il peut détecter tous les pointeurs inverses et les replacer dans la table. Il faut noter cependant que les objets en cours contiennent aussi des pointeurs inverses, mais il ne faut pas les placer dans la table, sous peine de faire déborder celle-ci. Toutes ces subtilités compliquent fortement le code du GC et détruisent la modularité du GM.

3.4 Performances et critique du GC hybride

3.4.1 GC mineur

Mesures

La figure 3.7 donne le résultat de mesures effectuées sur quelques programmes ML intéressants.

- Le *rendement* est la proportion d'objets inaccessibles dans le tas mineur au moment du déclenchement du GC mineur. Cela correspond à peu près au pourcentage du travail de désallocation qui est effectué par le GC mineur, le reste étant fait par le GC majeur.
- La *latence* est le temps de latence du GC mineur (on donne la moyenne et le maximum).
- Le *temps de calcul* est le temps de calcul consommé par le GC mineur, exprimé comme pourcentage du temps de calcul total du programme.

Les mesures ont été effectuées sur une DECstation 3000/400 munie d'un processeur Alpha cadencé à 133 MHz, ce qui donne une puissance de calcul d'environ 100 MIPS. Les programmes sont :

KB Une implémentation "naïve" de l'algorithme de complétion de Knuth-Bendix, avec un jeu de règles d'exemple standard (le jeu complet donné dans [67]).

compilateur Le compilateur Caml Light, compilant son propre code.

CoQ Le système CoQ (version 5.8), un système de preuve assistée par ordinateur, exécutant son jeu de tests étendu [38].

SIMPLE Un programme de simulation de mécanique des fluides qui effectue des calculs sur des matrices de nombres flottants. Ce programme est tiré de [5].

Les trois premiers sont des programmes ML assez typiques, qui manipulent des valeurs symboliques et utilisent peu d'affectations. Le dernier est intéressant, car il s'agit d'un programme FORTRAN de calcul numérique traduit en ML. Il représente donc un programme atypique du point de vue des taux d'allocation et de mutation.

Comme on le voit sur la figure 3.7, le GC mineur représente un coût assez faible par rapport au temps d'exécution des programmes. Le temps consacré au GC mineur est très rentable, puisqu'il désalloue la plupart des objets. De plus, son temps de latence est parfaitement acceptable pour des applications interactives. On peut facilement augmenter le rendement et diminuer le temps de calcul en augmentant la taille du tas mineur. En effet, le GC mineur est alors moins fréquent (puisqu'il y a plus d'allocations entre deux GC mineurs), et plus efficace (puisque les objets jeunes ont plus de temps pour mourir). Cependant, cette modification augmente aussi les temps de latence du GC mineur. Les mesures ont été effectuées avec un tas mineur de 128 kilo-octets.

D'autre part, des mesures effectuées sur le programme KB indiquent que le GC mineur passe la moitié de son temps à chercher les racines dans la pile. On peut donc améliorer notablement ses performances si on évite de parcourir toute la pile à chaque GC mineur. En effet, à la fin d'un cycle de GC mineur, le tas mineur est vide, donc il n'y a plus aucun pointeur vers lui. En particulier, il n'y a pas de pointeur vers le tas mineur dans la pile. Le GC mineur n'a donc pas besoin de chercher des racines dans la portion de la pile qui n'a pas changé depuis le cycle précédent, puisqu'elle ne contient aucun pointeur vers le tas mineur (donc aucune racine du GC mineur).

Il est relativement facile de détecter la portion de la pile qui n'a pas changé. En effet, le mutateur ne peut modifier que le sommet de la pile, donc on ne peut jamais avoir une portion inchangée au dessus d'une portion changée. La pile est donc divisée en deux parties contiguës : la partie inchangée (au fond) et la partie changée (au sommet). Il suffit donc que le collecteur puisse détecter la limite entre les deux.

La pile est utilisée par l'interpréteur de byte-code pour stocker des blocs d'appel de fonctions. Ces blocs sont des éléments qui sont empilés lors des appels de fonctions et dépilés lors du retour. Ils contiennent en général plusieurs racines chacun. Il suffit que chaque bloc d'appel contienne un drapeau (dont la signification est : "ce bloc a été empilé après le dernier GC mineur") et que le mutateur positionne ce drapeau à "vrai" quand il empile un bloc. Le collecteur positionne tous les drapeaux de la pile à "faux" quand il a fini son cycle. Ainsi, au début du cycle suivant, il peut reconnaître la portion inchangée de la pile : elle est constituée des blocs dont le drapeau est faux. Cette technique permet, pour un coût très faible, d'obtenir une bonne amélioration des performances du GC mineur. Elle peut s'appliquer à n'importe quel système de GC à générations.

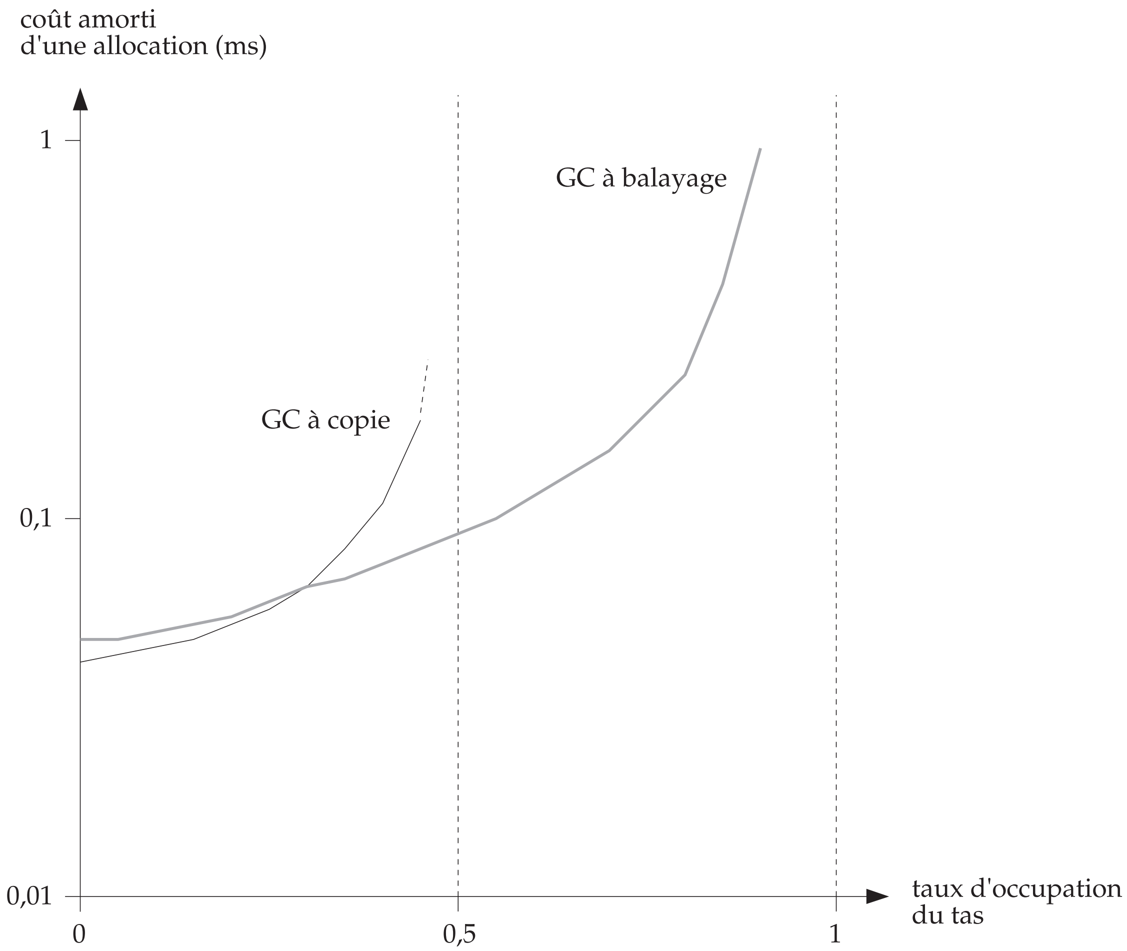


Figure 3.8: Coût des GC à copie et à balayage

Intérêt du GC mineur à copie

Le GC mineur effectue l'essentiel du travail du GM. Il est très efficace et son code est simple. Le surcoût qu'il impose à l'opération d'affectation est négligeable grâce au faible taux de mutation de ML. Un mécanisme de générations avec un GC mineur à copie s'avère très utile pour obtenir un GM efficace et fiable.

3.4.2 GC majeur

Cette section reprend les mesures présentées dans [35].

Mesures

La mesure de l'efficacité du GC majeur est le coût amorti de l'allocation d'un objet : le temps de calcul de la fonction d'allocation, plus la fraction du coût du GC que cette allocation a contribué à déclencher. L'efficacité est fonction du taux d'occupation du tas : la taille des objets vivants divisée par la taille totale du tas.

Les mesures ont été effectuées sur une station Sun 3/60 avec un mutateur qui alloue alternativement dans deux listes des petits objets de tailles aléatoires, et qui rend inaccessibles les deux listes alternativement. L'utilisation de deux listes cause une fragmentation. Le temps calculé est le coût amorti de l'allocation, c'est-à-dire le coût de l'allocation elle-même, plus la fraction du coût du GC qu'elle a contribué à déclencher. Malheureusement, il n'a pas été possible de séparer le temps de calcul du mutateur lui-même, ce qui rend les mesures peu précises dans les cas où le coût du GC est faible.

Les GC à copie et à balayage ont un degré de liberté : la taille du tas. C'est le GM qui décide de changer la taille du tas en fonction du comportement du mutateur. Or le GC est plus performant quand le tas est presque vide, comme on le voit sur la figure 3.8. Le GM peut toujours réduire le coût amorti de l'allocation en augmentant la taille du tas, mais le temps de calcul n'est pas la seule mesure pertinente : la taille de la mémoire occupée par le processus ne doit pas devenir trop grande car le système de mémoire de la machine (caches, TLB, mémoire virtuelle) est optimisé pour des processus de taille raisonnable. En pratique, plus un processus utilise de mémoire, plus les temps moyens d'accès à la mémoire sont grands.

Un programme donné utilise un certain nombre d'objets vivants, la taille du tas et son taux d'occupation sont donc inversement proportionnels. On peut donc considérer que le GM choisit le taux d'occupation du tas, non seulement en fonction du coût du GC, mais aussi en fonction de facteurs extérieurs comme la taille maximum tolérée par l'utilisateur. Notons aussi que le temps de latence augmente avec la taille du tas dans le cas du GC à balayage, mais pas pour le GC à copie.

On remarque sur la figure 3.8 que le GC à copie est intéressant pour les taux d'occupation inférieurs à 1/3. Au delà, à taux d'occupation égal, il est plus coûteux en temps de calcul. On peut aussi dire que, pour un temps de calcul donné, il est plus coûteux en mémoire, puisqu'il demande un taux d'occupation plus faible, donc un tas plus grand.

Le GC hybride a un deuxième degré de liberté : la proportion de balayage. La figure 3.9 donne le coût amorti de l'allocation en fonction des deux paramètres choisis par le GM : proportion de balayage et taux d'occupation du tas. L'extrémité gauche des courbes correspond au fonctionnement en copie pure. Les performances sont pratiquement identiques celles du GC à copie de la figure 3.8. L'extrémité droite des courbes correspond au fonctionnement en balayage pur. Les performances sont très proche de la courbe du GC à balayage de la figure 3.8.

On remarque un effet intéressant pour les taux d'occupation élevés : le coût minimal est atteint pour un certain compromis entre copie et balayage. Le GC hybride est donc plus efficace que chacune de ses deux composantes : il ne représente pas seulement des niveaux intermédiaires de performances. Le GC hybride est plus efficace que le GC à copie car il réserve moins de mémoire, donc il se déclenche moins souvent. Il est plus efficace que le GC à balayage pour deux raisons : d'une part il ne balaye pas tout le tas, et d'autre part il compacte la mémoire, ce qui donne une allocation plus efficace (car

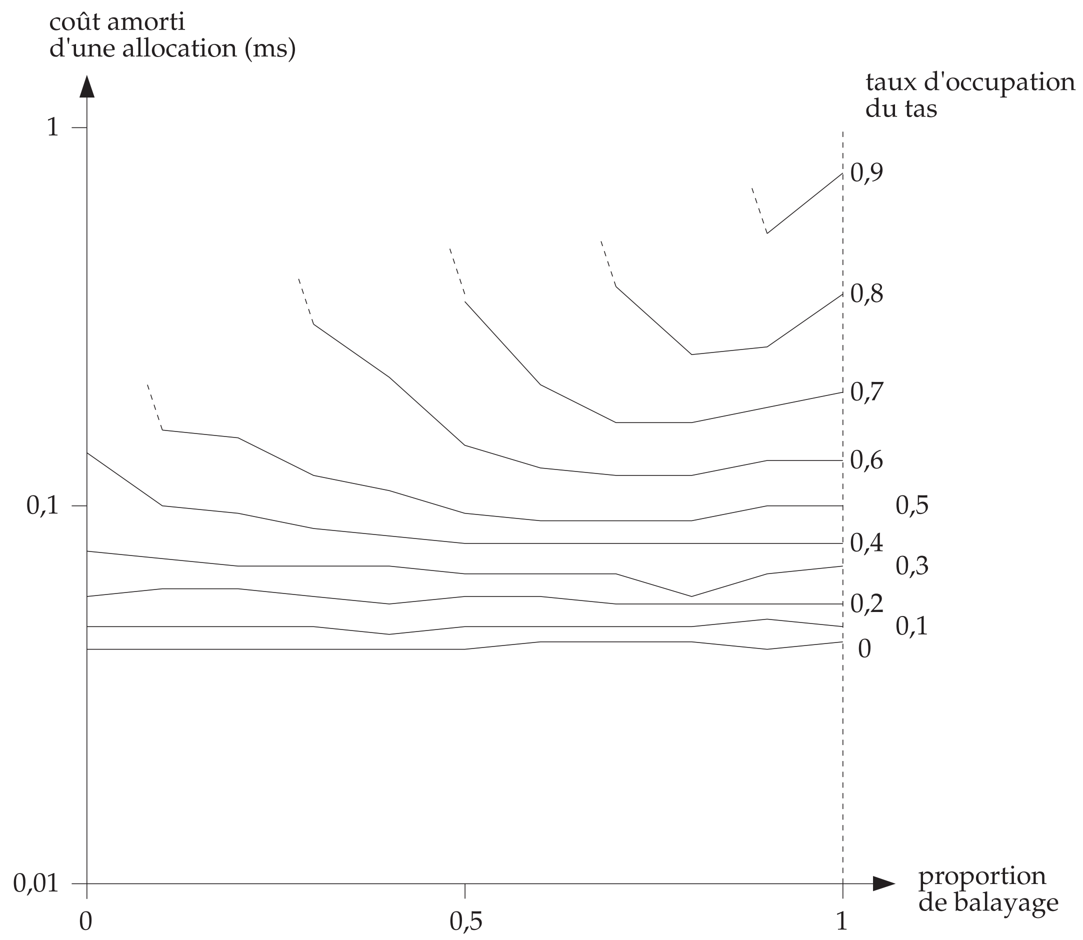


Figure 3.9: Coût du GC de Lang et Dupont

on trouve plus vite un bloc de taille suffisante si la liste libre est moins fragmentée). Comme on le voit sur la figure 3.9, l'optimum est atteint lorsque l'on réserve environ la moitié de la mémoire libre pour l'espace d'arrivée.

Intérêt du GC hybride

Le GC hybride a de bonnes performances par rapport aux GC à copie ou à balayage, et son degré de liberté supplémentaire (la proportion de balayage) lui permet de s'adapter à toutes les situations.

Cependant, le gain en performances apporté par l'algorithme de Lang et Dupont est complètement masqué par le GC à générations : le GC majeur ne représente que 1% du temps de calcul du programme, donc ses performances ont peu d'importance (un GC deux fois plus lent ne ralentit le programme que de 1%). Le véritable intérêt du GC hybride réside dans son action de compactage de la mémoire. Il garantit une fragmentation limitée, alors que dans un GC à balayage la fragmentation est un phénomène aléatoire qui peut devenir grave et augmenter arbitrairement le coût de l'allocation. Malheureusement, nous n'avons pas effectué de mesures de la fragmentation, et il est difficile de connaître son impact sur le coût de l'allocation.

Comme il n'est pas incrémental, le GC majeur a un temps de latence important, et qui augmente avec la taille du tas. Ce problème le rend difficile à utiliser dans des programmes interactifs (c'est-à-dire quasi-temps-réel).

Du point de vue de l'implémentation, le tas contigu est une contrainte importante qui nuit à la portabilité du système à cause de l'incompatibilité avec `malloc`. Ce problème pourrait être résolu en utilisant la même technique de table de pages que dans le chapitre 7, au prix d'une complexité accrue et de performances légèrement moins bonnes.

Enfin, la mise au point de ce GC a été longue et délicate, et il est difficile d'avoir pleine confiance en sa fiabilité, même après des années de fonctionnement sans problème.

Chapitre 4

Algorithme concurrent à balayage

Ce chapitre présente notre algorithme de GC concurrent à balayage pour machines parallèles à mémoire partagée. Il s'agit à notre connaissance du premier algorithme de GC concurrent qui réunisse les caractéristiques suivantes :

- Il fait l'objet d'une preuve formelle de correction.
- Il utilise un modèle réaliste de la machine, et il utilise très peu de primitives de synchronisation coûteuses.
- Il peut fonctionner avec un mutateur multi-tâches, et il n'impose pas de synchronisation parasite entre les tâches du mutateur.
- Il est utilisé dans l'implémentation d'un GM efficace (qui est décrit dans le chapitre 7).

Ce chapitre est organisé comme suit. La section 4.1 décrit le type de machines parallèles auquel on s'intéresse, expose les caractéristiques minimales d'un GC concurrent pour ces machines et présente les solutions trouvées dans la littérature. La section 4.2 présente l'algorithme de base de [33]. La section 4.3 explique par une série de scénarios pourquoi l'algorithme de base ne s'adapte pas directement à un modèle réaliste de machine. Enfin, dans la section 4.4, nous exposons le fonctionnement de notre algorithme.

4.1 Le problème du GC concurrent

4.1.1 Machines multi-processeurs à mémoire partagée

On considère une implémentation de ML (ou de tout autre langage à fort taux d'allocation et faible taux de mutation) sur une machine multi-processeurs à mémoire partagée. Pour exploiter la puissance de la machine, une telle implémentation doit

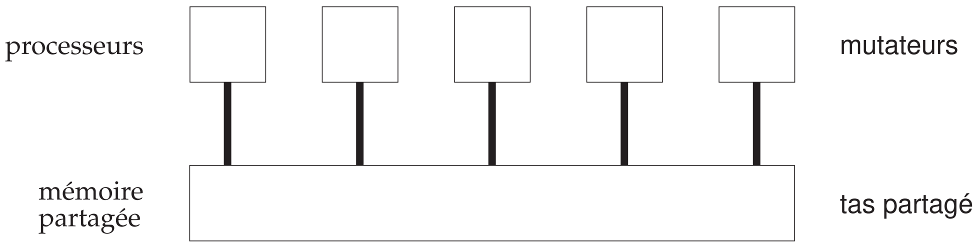


Figure 4.1: Machine parallèle simpliste

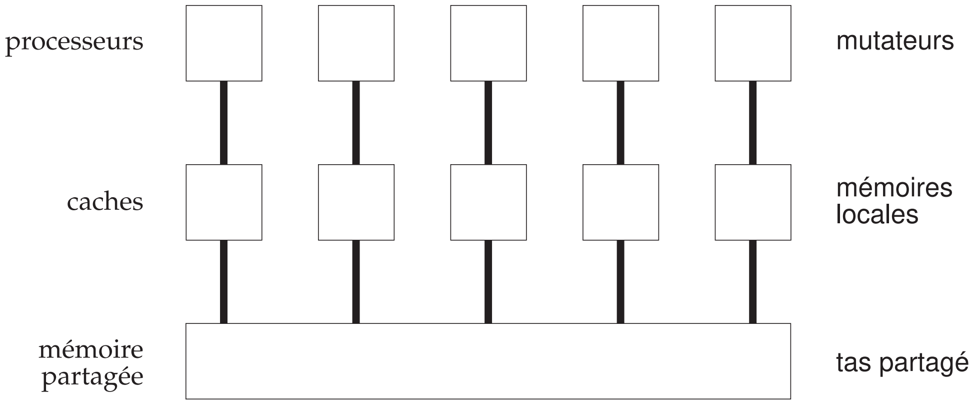


Figure 4.2: Machine parallèle réaliste

permettre d'écrire des programmes multi-tâches. Cela peut se faire explicitement, en écrivant des programmes parallèles dans une extension parallèle de ML, ou bien implicitement, en utilisant un compilateur qui parallélise automatiquement les programmes. Dans les deux cas, il faut un système de gestion de mémoire adapté à ce contexte multi-tâches.

Mémoire partagée et mémoires locales

La figure 4.1 illustre le type de machines auquel on s'intéresse : plusieurs processeurs travaillent en même temps et accèdent à une mémoire commune. L'organisation des programmes reflète cette architecture : plusieurs processus travaillent simultanément sur un tas commun. Du point de vue du gestionnaire de mémoire, on considère que le programme est composé de plusieurs mutateurs.

Malheureusement, cette vision d'une machine parallèle est trop simplifiée. Dans une machine fonctionnant suivant ce modèle, la mémoire devrait répondre simultanément aux requêtes de lecture et d'écriture de tous les processeurs. Or les microprocesseurs modernes sont si rapides que les circuits de mémoire ne peuvent même pas répondre assez vite aux requêtes d'un seul processeur.

Dans les systèmes mono-processeur, on utilise un cache : une mémoire de petite taille mais rapide, où sont stockées les données fréquemment utilisées. Le cache est entièrement géré par le matériel : un circuit électronique choisit certaines adresses de la mémoire et intercepte les lectures et écritures faites par le processeur à ces adresses, pour les transformer en lectures et écritures dans le cache. Ces adresses sont choisies (et peuvent changer) dynamiquement en fonction des lectures et des écritures faites par le processeur. La gestion du cache est entièrement transparente : du point de vue du logiciel, la seule différence par rapport à une machine sans cache est que l'accès aux données fréquemment utilisées est beaucoup plus rapide.

De même, dans les systèmes multi-processeurs chaque processeur est doté d'un cache qui contient les données utilisées fréquemment par ce processeur. On diminue ainsi la fréquence des accès à la mémoire partagée. D'où le modèle réaliste de machine parallèle de la figure 4.2.

Là aussi, la gestion des caches est entièrement transparente : le logiciel garde une vision de la machine conforme à la figure 4.1. Si un processeur tente d'accéder à une donnée qui se trouve dans le cache d'un autre processeur, les circuits de gestion des caches devront faire transiter cette donnée par la mémoire partagée pour la transférer d'un processeur à l'autre. Ce déplacement d'une donnée est au moins 10 à 20 fois plus lent qu'une simple lecture ou écriture dans le cache.

Pour obtenir des programmes efficaces, le programmeur (ou le compilateur) doit donc tenir compte de la présence des caches : chaque tâche doit travailler autant que possible sur des données qu'elle est la seule à utiliser, et qui resteront donc dans le cache du processeur qui exécute cette tâche.

L'organisation du programme reflète alors celle de la machine : chaque tâche du programme se réserve une zone de mémoire privée, que nous appellerons *mémoire locale*, dans laquelle elle effectue l'essentiel de ses calculs. Cette mémoire locale contient au moins la pile d'exécution du processus. On peut aussi y placer la zone mineure d'un GC à générations, comme on le verra au chapitre 7.

Les autres tâches du programme, y compris la tâche chargée du GC, ne doivent pas accéder à la mémoire locale d'une tâche, sous peine d'annuler l'avantage apporté par la présence de caches, donc de ralentir excessivement le déroulement du programme.

Séquentialisation des accès à la mémoire

Dans une machine parallèle à mémoire partagée, il n'y a pas d'accès simultanés à la mémoire car la mémoire ne peut répondre qu'à une requête à la fois. Lorsque deux requêtes arrivent simultanément, la mémoire en choisit une à effectuer avant l'autre. On dit que la mémoire *séquentialise* les accès. Pour un observateur qui examine le contenu de la mémoire et son évolution au cours du temps, le programme n'est pas parallèle : les modifications (et les lectures) de la mémoire par le programme se déroulent en séquence. La différence avec un programme séquentiel se trouve dans le non-déterminisme : le même programme parallèle peut donner plusieurs séquences différentes d'accès à la mémoire, selon les choix effectués dynamiquement par la mémoire, tandis qu'un programme séquentiel se déroule toujours de la même façon.

Une instruction ou séquence d'instructions est dite *atomique* si ses accès à la mémoire ne peuvent pas être entrelacés avec ceux d'un autre processus. Autrement dit, les lectures et écritures d'une instruction atomique se suivent sans interruption dans toutes les séquentialisations possibles du programme : aucun autre accès à la mémoire ne peut avoir lieu entre deux accès d'une instruction atomique.

Par exemple, une lecture est atomique, car elle ne fait qu'un accès à la mémoire. L'incrémentation d'une variable n'est normalement pas atomique, car elle est composée de deux instructions, lire l'ancienne valeur et écrire la nouvelle, et les instructions d'un autre processus peuvent se glisser entre ces deux instructions. Pour les raisons expliquées dans la section suivante, les processeurs fournissent souvent une instruction atomique d'incrémentation. Cette instruction effectue une lecture et une écriture et elle empêche la mémoire d'accepter d'autres requêtes entre les deux.

Primitives de synchronisation

Tous les programmes parallèles non triviaux utilisent des primitives de synchronisation telles que sémaphores, verrous, etc. Ces primitives permettent de résoudre le problème de l'exclusion mutuelle : lorsqu'un processus veut s'assurer qu'il est le seul à travailler sur une structure de données partagée, il prend le verrou associé à cette structure, ce qui empêche les autres processus de prendre le verrou pour travailler sur la même structure. Lorsqu'il en a fini avec cette structure, le processus relâche le verrou, ce qui permet aux autres tâches d'accéder à cette structure.

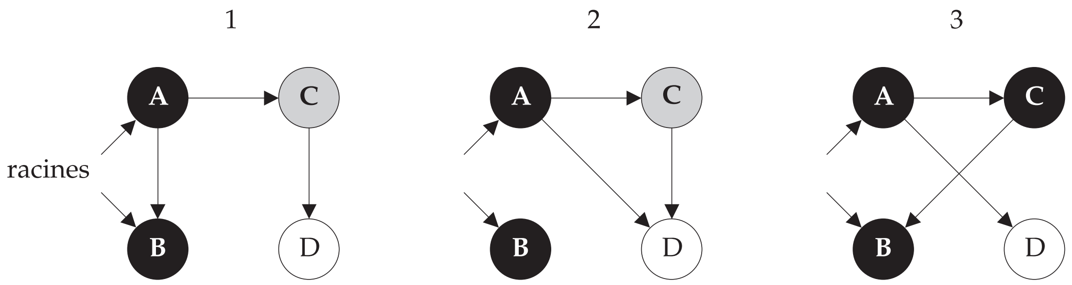
Toutes les machines multi-processeurs fournissent des primitives de synchronisation. En théorie, il est possible de les implémenter sans aide du matériel, uniquement par programme [31, 78]. En pratique c'est trop difficile et inefficace, et le matériel fournit une instruction de synchronisation élémentaire grâce à laquelle on implémente les mécanismes de synchronisation de haut niveau tels que les sémaphores et les verrous.

Ces mécanismes de synchronisation de haut niveau sont généralement fournis au programmeur sous forme de fonctions de la bibliothèque standard ou d'appels système. Leur coût est beaucoup plus grand qu'une simple lecture ou écriture de la mémoire ; il est donc préférable de les utiliser avec parcimonie. En particulier, il n'est pas réaliste pour le GM d'imposer aux processus de prendre un verrou pour chaque accès à la mémoire.

4.1.2 Caractéristiques d'un bon GC concurrent

Comme il est noté dans [33], un GC concurrent ne peut pas fonctionner sans la coopération du mutateur. En effet, si le mutateur modifie le graphe mémoire pendant que le collecteur le parcourt, certains objets peuvent échapper à l'attention du collecteur et rester non-marqués bien qu'ils soient encore accessibles. Ce problème est illustré par la figure 4.3. On utilise un marquage à trois couleurs (cf. section 2.3.2).

Le problème dans ce scénario est la combinaison de deux actions du mutateur : il efface le pointeur de l'objet gris vers le blanc et il crée un pointeur de l'objet noir vers le blanc. La suppression du pointeur gris $\to$ blanc empêche le collecteur de marquer l'objet blanc, et la création du pointeur noir $\to$ blanc permet à l'objet de rester accessible. Pour éviter ce scénario, il faut imposer une action supplémentaire au mutateur



Le collecteur marque C et parcourt A et B (1)
Le mutateur remplace dans A le pointeur sur B par un pointeur sur D (2)
remplace dans C le pointeur sur D par un pointeur sur B
Le collecteur termine son marquage et désalloue D qui est encore accessible (3)

Figure 4.3: La coopération du mutateur est indispensable au GC concurrent

lors de l'affectation. Il y a deux solutions : la première est que le mutateur marque en gris (s'il est blanc) le nouvel objet, c'est-à-dire celui qui est au bout du pointeur créé par l'affectation (dans notre exemple le mutateur marquerait D lors de la première affectation). La deuxième solution est que le mutateur marque en gris l'ancien objet, c'est-à-dire celui qui est au bout du pointeur effacé par l'affectation (le mutateur marquerait D lors de la deuxième affectation).

Dans la suite de ce chapitre, nous appellerons *écriture* (ou *affectation*) l'opération d'affectation directe dans un mot de la mémoire ou dans un registre, et *modification* la procédure du mutateur qui change le graphe mémoire. Cette procédure est fournie par le GM et elle effectue non seulement l'écriture (dans un objet du tas) mais aussi les actions nécessaires pour permettre au GC de fonctionner malgré ce changement du graphe mémoire (la nature précise de ces actions dépend de l'algorithme de GC utilisé). La figure 4.3 montre donc que la modification ne peut pas se faire par simple écriture.

La figure 4.4 représente le graphe mémoire dans une machine à mémoire partagée. Les racines du graphe mémoire sont tous les pointeurs qui viennent des mémoires locales et qui pointent dans le tas, et les variables globales. Les variables globales sont les seules racines directement connues du collecteur ; les autres racines sont stockées dans les mémoires locale des mutateurs, auxquelles le GC n'a pas accès. Il faut donc qu'il puisse demander à chaque mutateur de lui donner ses racines. Plus généralement, selon l'algorithme utilisé, les mutateurs devront effectuer une partie du travail du GC.

Les instructions exécutées par les mutateurs n'ont d'incidence sur la gestion de la mémoire que si elles manipulent des pointeurs ou si elles servent à coopérer avec le GC. Nous distinguerons sept classes d'actions intéressantes du point de vue de la gestion de la mémoire :

manipulation de la mémoire locale : duplication ou suppression de pointeurs (vers le tas) contenus dans la mémoire locale,

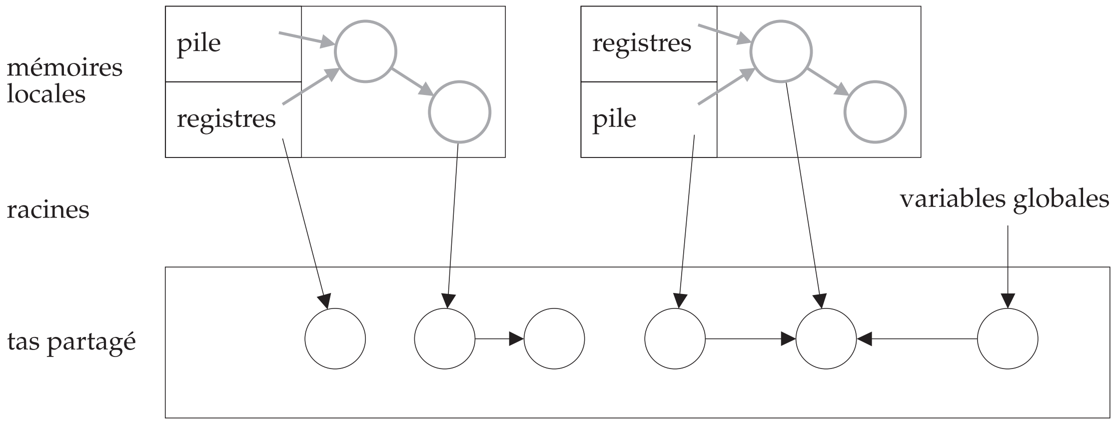


Figure 4.4: Le graphe mémoire du GC concurrent. Les objets représentés en gris ne sont pas gérés par le GC concurrent. Seuls les objets et les pointeurs en noir font partie du graphe mémoire.

lecture d'un pointeur contenu dans un objet pour le mettre dans un registre ou dans la mémoire locale,

réservation de mémoire libre en vue de créer de nouveaux objets,

création d'un objet dans la mémoire réservée,

remplissage d'un objet nouvellement créé,

modification du graphe mémoire,

coopération avec le collecteur (en particulier, marquage des objets-racines). Cette catégorie regroupe toutes les opérations effectuées par les mutateurs pour le compte du collecteur.

L'opération d'allocation est divisée en trois actions élémentaires : **réservation**, **création** et **remplissage**. La **réservation** est nécessairement coûteuse. En effet, les demandes d'allocation des différents mutateurs entrent en concurrence pour obtenir la mémoire libre fournie par le GM. L'opération d'allocation doit donc utiliser une primitive de synchronisation, car la liste libre est une structure de données dont l'accès doit se faire en exclusion mutuelle. Or les primitives de synchronisation sont coûteuses. En séparant **réservation** et **création**, on peut amortir le coût de cette synchronisation en gardant une liste libre locale à chaque mutateur, qui est remplie en une action coûteuse (**réservation**) à partir de la liste libre partagée, puis utilisée pour plusieurs allocations (**création**).

Les actions de **remplissage** et de **modification** correspondent à la même opération : une écriture dans la mémoire partagée. Il est utile de les distinguer car elles ont des fréquences et des conditions d'utilisation différentes. Pour une action de **remplissage**, l'objet modifié n'est accessible que par le mutateur qui l'a créé (ML ne permet pas d'allouer un objet non initialisé, donc les mutateurs doivent toujours remplir complètement un objet juste après sa création). En revanche, une opération de **modification** change le contenu d'un objet qui peut être partagé. Comme expliqué au début de cette section, le mutateur ne peut pas se contenter de modifier l'objet : il doit signaler cette modification au GC d'une façon ou d'une autre. L'action de **modification** est donc plus difficile à implémenter que le **remplissage**. Heureusement, elle est beaucoup moins fréquente que le **remplissage** dans les programmes ML.

L'action de **coopération** existe parce que les mutateurs ont une mémoire locale à laquelle le collecteur ne peut pas accéder directement. Les mutateurs doivent donc effectuer certaines opérations pour le compte du GC, notamment marquer leurs racines au début du cycle de GC. Ces opérations sont regroupées dans l'action de **coopération**.

Un algorithme de GC concurrent économe doit remplir les conditions suivantes :

1. Les actions de **manipulation**, **lecture** et **remplissage** sont exécutées très souvent par le mutateur. Le GC ne doit leur imposer aucun surcoût.

2. Le mutateur ne doit être obligé d'utiliser les primitives de synchronisation que pour les actions de **réservation**, pour lesquelles la synchronisation est inévitable.

3. Le GC doit laisser une grande latitude aux mutateurs quant au moment où ils exécutent les actions de **coopération**.

4. Le coût total des actions de **coopération** par mutateur et par cycle de GC doit être borné.

5. Un cycle de GC doit toujours se terminer, même si on agrandit le tas et si on crée des processus.

La condition 1 est une contrainte d'efficacité indispensable. En effet, une action de **manipulation**, **lecture** ou **remplissage** est implémentée par une instruction de la machine. Si on impose une opération supplémentaire lors de ces actions, elle devra pour être utile comporter au minimum un accès à la mémoire partagée, qui est aussi coûteux que l'action de **lecture** ou de **remplissage** elle-même, et qui peut être jusqu'à 20 fois plus coûteux qu'une **manipulation** locale. Un tel ralentissement par rapport à une version séquentielle du même système est inacceptable.

La condition 2 est une conséquence du coût de la synchronisation. On accepte pour les opérations de **création**, **modification** et **coopération** un surcoût modéré (par exemple, quelques accès supplémentaires à la mémoire et un ou deux tests), car ces opérations sont peu fréquentes, mais l'utilisation de primitives de synchronisation représenterait un surcoût trop important.

La condition 3 implique qu'un mutateur n'a pas besoin de se tenir prêt à coopérer avec le GC à tout instant. Ceci autorise les mutateurs à avoir des états transitoires, dans lesquels la coopération n'est pas possible, ce qui est indispensable pour des raisons d'efficacité. Cette condition permet aussi d'écrire des programmes temps-réel : un mutateur peut suspendre la coopération avec le GC le temps d'exécuter une portion de code dont la vitesse est critique.

La condition 4 impose une borne sur la quantité de travail que le mutateur doit effectuer pour le compte du collecteur avant d'obtenir de la mémoire en retour.

La condition 5 est simplement le reflet du fait que les ressources de la machine ne sont pas bornées. En théorie, elles sont bornées, mais un programme normal ne les épuise jamais. Avec un système d'exploitation multi-utilisateurs tel que Unix, un processus qui épuise la mémoire, par exemple, rend la machine inutilisable pour les autres utilisateurs. Un tel comportement est acceptable si les mutateurs utilisent toute la mémoire pour faire du travail utile, mais pas s'il est causé par un collecteur qui attend que les ressources soient épuisées pour faire son travail.

En ce qui concerne l'efficacité du collecteur lui-même, nous imposons des contraintes beaucoup plus faibles : les actions de marquage et de désallocation d'un objet ne doivent pas exiger de synchronisation, et le travail total du collecteur doit être proportionnel à la taille du tas et au nombre de mutateurs actifs, du moins en l'absence d'opérations de **modification**, qui sont peu fréquentes. La vitesse du collecteur doit être suffisante pour désallouer les objets aussi vite qu'ils sont alloués par les mutateurs. En pratique, on constate que la vitesse du collecteur ne pose pas de problème, surtout si le GC concurrent est complété par un mécanisme de générations comme celui décrit dans le chapitre 7.

4.1.3 Les algorithmes existants et leurs défauts

Cette section passe en revue une sélection d'algorithmes tirés de la littérature et explique en quoi ils violent tous au moins une des conditions 1–5.

GC concurrents à copie

Nos contraintes semblent élémentaires, mais elles interdisent pratiquement l'utilisation d'un GC à copie. En effet, celui-ci doit déplacer les objets. Il ne peut pas accéder la mémoire locale des mutateurs pour mettre à jour les pointeurs vers les objets déplacés, donc les mutateurs eux-mêmes doivent s'en charger, ce qui viole toujours au moins l'une des contraintes 1–3, comme expliqué ci-après.

L'algorithme de Baker [11] et ses dérivés [21, 46, 48, 77] ajoutent un test, une indirection supplémentaire ou même une synchronisation sur l'action de **lecture**, et donc ne respectent pas la contrainte 1.

D'autres algorithmes [28, 29, 27, 74, 86] utilisent la protection de pages de la mémoire virtuelle pour éviter le test (en le faisant faire directement par le matériel), mais ils ne respectent pas les contraintes 2 et 3 : les mutateurs sont interrompus pour des temps assez longs par les défauts de page, qui surviennent à des instants imprévisibles.

Enfin, des algorithmes intéressants ont été proposés récemment [20, 76], mais ils exigent un rendez-vous global de tous les mutateurs à chaque cycle de GC, ce qui est contraire à 3 ou à 2.

GC concurrents à balayage

L'algorithme de base pour le GC concurrent à balayage est [33]. Cet algorithme ne tient pas compte de la mémoire locale : il suppose que les pointeurs locaux des mutateurs ne pointent que vers des objets accessibles par ailleurs, ce qui ne peut être garanti qu'en imposant à chaque manipulation locale de pointeurs d'être reflétée dans la mémoire partagée, ce qui est contraire à 1. Les dérivés [15, 44, 54] de l'algorithme de base souffrent du même défaut. De plus, ces algorithmes ne fonctionnent qu'avec un seul mutateur. La version multi-mutateurs [57] exige explicitement une synchronisation pour l'opération de **modification**.

Les autres algorithmes proposés imposent un test ou une synchronisation sur la **lecture** [26, 88], en contradiction avec 1, ou une synchronisation sur la **modification** [29, 79], ou même sur la **manipulation** [49], ce qui viole 2.

4.2 L'algorithme de base

Cette section décrit brièvement l'algorithme de [33], qui est à la base de notre algorithme, et que nous appellerons l'algorithme de base. L'algorithme de base fonctionne avec un seul mutateur. La section suivante expose les problèmes que nous avons rencontrés et résolus en adaptant cet algorithme à plusieurs mutateurs et à notre modèle de machine réaliste.

Le tas est un tableau (de taille fixe) d'objets ; chaque objet a un nombre fixe de champs. On a donc les déclarations :

$$\begin{array}{lll} \textbf{const} & \mathit{Top}, \mathit{MaxIndex} \in \mathbb{N} & \\ \textbf{type} & \mathit{ADDR} & \equiv \{0, \ldots, \mathit{Top}-1\} \\ & \mathit{INDEX} & \equiv \{0, \ldots, \mathit{MaxIndex}\} \\ \textbf{var} & \mathit{heap} & \in \textbf{array } [\mathit{ADDR}, \mathit{INDEX}] \textbf{ of } \mathit{ADDR} \end{array}$$

Il y a un nombre fixe de variables globales. Toutes les racines sont dans ces variables globales. L'une d'elles est la tête de la liste libre, qui est une liste chainée.

$$\textbf{const} \quad \mathit{Globals} \in \textbf{set of } \mathit{ADDR}$$

La couleur des objets est stockée dans un tableau, les objets sont tous blancs au début de l'exécution du programme.

$$\begin{array}{ll} \textbf{type} & \mathit{COLOR} \equiv \{\mathit{White}, \mathit{Gray}, \mathit{Black}\} \\ \textbf{var} & \mathit{color} \quad \in \textbf{array } [\mathit{ADDR}] \textbf{ of } \mathit{COLOR} \\ \textbf{init} & \forall x \in \mathit{ADDR}, \mathit{color}[x] = \mathit{White} \end{array}$$

Les couleurs sont utilisées comme dans un GC à balayage normal. Comme la tête de la liste libre est une variable globale, la liste libre est considérée comme accessible, et l'allocation est un cas particulier de la **modification**. En fait, le **remplissage**, la **réservation** et la **création** sont des cas particuliers de la **modification**, qui est implémentée par la procédure *Update*. Le mutateur est libre d'appeler cette

$Mark$:
  foreach $x \in Globals$ **do** $MarkGray(x)$

$Scan$:
  var $dirty \in BOOL$
  repeat
    $dirty \leftarrow$ **false**
    foreach $x \in ADDR$ **do**
      if $color[x] = Gray$ **then**
        $dirty \leftarrow$ **true**
        foreach $i \in INDEX$ **do** $MarkGray(heap[x, i])$
        $color[x] \leftarrow Black$
  until $\neg\, dirty$

$Sweep$:
  foreach $x \in ADDR$ **do**
    if $color[x] = White$ **then** *ajouter* x *à la liste libre*

$Clear$:
  foreach $x \in ADDR$ **do**
    $color[x] \leftarrow White$

Figure 4.5: Le collecteur de l'algorithme de base

procédure à tout moment, en lui donnant comme arguments deux objets accessibles et un index valide. Notre modèle du mutateur est donc un processus qui exécute la procédure *Update* à sa discrétion ; les autres calculs effectués par le mutateur n'ayant pas d'influence sur la gestion de la mémoire. Le code de la procédure *Update* est le suivant :

$MarkGray(x) \equiv$
  if $color[x] = White$ **then** $color[x] \leftarrow Gray$

$Update(x, i, y) \equiv$
  $heap[x, i] \leftarrow y$
  $MarkGray(y)$

Le collecteur est un processus séquentiel qui travaille en parallèle avec le mutateur. Son code est donné par la figure 4.5. Le cycle du collecteur est divisé en quatre étapes :

Mark Marque (en gris) les objets-racines.

Scan Parcourt le tas et noircit tous les objets vivants.

Sweep Désalloue tous les objets blancs en les insérant dans la liste libre.

Clear Blanchit tous les objets, préparant ainsi le prochain cycle de GC.

L'étape *Scan* est un marquage à trois couleurs classique, comme expliqué dans la section 2.3.2. On utilise une variable locale *dirty* pour savoir s'il reste des objets gris dans le tas. Cette variable vaut **false** à la fin du parcours de recherche des objets gris si le GC est certain qu'il ne reste pas d'objet gris. Si elle vaut **true**, il reste peut-être des objets gris.
Les deux invariants utilisés dans [33] pour prouver cet algorithme sont :

- Pendant l'étape *Scan*, tout objet blanc accessible est accessible depuis un objet gris par un chemin qui ne passe que par des objets blancs.
- Au début de l'étape *Sweep*, il n'y a pas d'objet gris.

De ces invariants, on déduit que tous les objets accessibles sont noirs au début de l'étape *Sweep*. Le deuxième invariant est facile à prouver si on suppose que *MarkGray* est atomique. On prouve d'abord que *MarkGray* ne peut pas marquer un objet en gris s'il ne reste pas au moins un autre objet gris dans le tas. En effet, l'objet x est accessible ; s'il est blanc, il doit rester au moins un autre objet gris dans le tas (à cause du premier invariant) ; s'il n'est pas blanc, *MarkGray* ne change pas sa couleur.

On prouve ensuite que la boucle **repeat** de *Scan* ne peut terminer que si le tas ne contient pas d'objet gris : la boucle **foreach** $x \in ADDR$ ne peut terminer sans positionner *dirty* que s'il n'y avait pas d'objet gris au moment où elle a commencé. En effet, un objet gris restera gris jusqu'à ce que le collecteur le noircisse. S'il n'y avait pas d'objet gris au moment où la boucle **foreach** a commencé, le mutateur n'a pas pu en créer, puisque *MarkGray* ne peut créer un nouvel objet gris que s'il en existe déjà un autre.

Pour le premier invariant, il faut prouver qu'il y a au plus un seul pointeur d'un objet noir vers un objet blanc : le pointeur de x vers y quand le mutateur est en train d'exécuter *Update* et qu'il a exécuté la première instruction mais pas encore la deuxième.

Pour prouver que la boucle de *Scan* termine, il suffit de remarquer que le nombre des objets accessibles blancs ou gris ne peut que diminuer : le mutateur ne change jamais la couleur d'un objet noir ou gris, et le collecteur noircit au moins un objet gris par tour de la boucle de *Scan*. Lorsqu'il noircit le dernier objet gris, le collecteur grise forcément au moins un objet blanc accessible, s'il en reste. Le nombre d'objets blancs accessibles doit donc finalement tomber à zéro, puis le nombre d'objets gris, et l'étape *Scan* devra alors se terminer.

On trouvera le détail de cette preuve dans [33]. L'algorithme reste correct si *MarkGray* n'est pas atomique, mais la preuve est plus compliquée.

Cet algorithme est sujet au problème des *objets flottants* : ce sont des objets qui deviennent inaccessibles pendant un cycle de GC et qui ne seront désalloués que par le cycle suivant. Pour être sûr de désallouer tous les objets qui sont inaccessibles à un instant donné, il faut donc attendre deux cycles de GC à partir de cet instant.

4.3 Adaptation à la machine réaliste

4.3.1 Problèmes dus aux mémoires locales

L'algorithme de base ne permet pas au mutateur d'avoir une mémoire locale : celui-ci doit s'assurer que ses variables locales pointent toujours vers des objets accessibles depuis les variables globales. Cela signifie non seulement que toutes les variables locales des mutateurs doivent rester visibles par le collecteur à tout moment (au moins sous forme d'une copie stockée dans les variables globales), mais aussi que toutes les manipulations de ces variables locales doivent payer le coût d'un appel à *MarkGray*. L'algorithme de base ne respecte donc pas la condition 1.

Pour résoudre ce problème, nous considérerons que les données locales de chaque mutateur (les racines de ce mutateur) sont inaccessibles au collecteur. Nous ajoutons une *poignée de mains* entre le collecteur et chaque mutateur : le collecteur demande aux mutateurs de colorier leurs objets-racines en gris, puis il attend que les mutateurs aient tous répondu. Les mutateurs devront exécuter de temps en temps une procédure *Cooperate* qui gère le coloriage des racines et la poignée de mains. Nous utiliserons la même procédure *Update* que dans l'algorithme de base.

L'implémentation de la poignée de mains se fait à l'aide d'une variable globale $status_C$ pour le collecteur, et une variable $status_m$ pour chaque mutateur m. Le collecteur change $status_C$ pour demander les racines (en alternant entre deux constantes $Sync_1$ et $Sync_2$). La procédure *Cooperate* d'un mutateur m compare $status_m$ et $status_C$. S'ils sont différents, le collecteur a demandé les racines. Pour répondre, *Cooperate* marque les racines de m puis met à jour $status_m$:

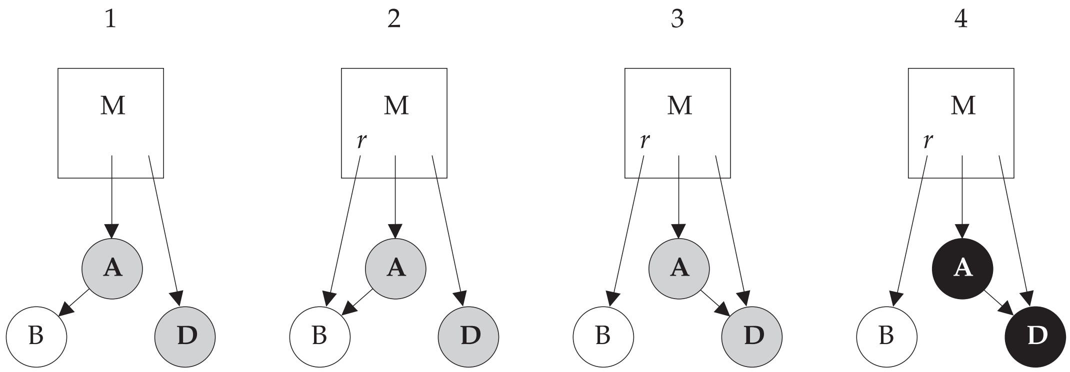
Cooperate $\equiv$
  if $status_m \neq status_C$ **then**
    Appeler MarkGray sur chaque racine.
    $status_m \leftarrow status_C$

Le collecteur effectue la poignée de mains lors de son étape *Mark* :

Mark :
  Changer $status_C$
  foreach $x \in Globals$ **do** $MarkGray(x)$
  Attendre $(\forall m, status_m = status_C)$

Le mutateur est libre de modifier sa mémoire locale à tout instant[1] sans en aviser le collecteur (à cause de notre contrainte 1). On peut donc avoir le scénario de la figure 4.6, dans lequel C désigne le collecteur, M le mutateur et r une variable locale de M. Dans ce scénario, le mutateur répond à la poignée de mains, puis place un pointeur sur un objet blanc (B) dans sa racine r. Enfin il coupe tous les autres pointeurs vers B. B est toujours accessible (il est pointé par r) mais le GC va le désallouer (car il n'a pas connaissance de r).

[1] Sauf bien sûr pendant qu'il exécute *MarkGray* ou *Cooperate* : le mutateur est un processus séquentiel.

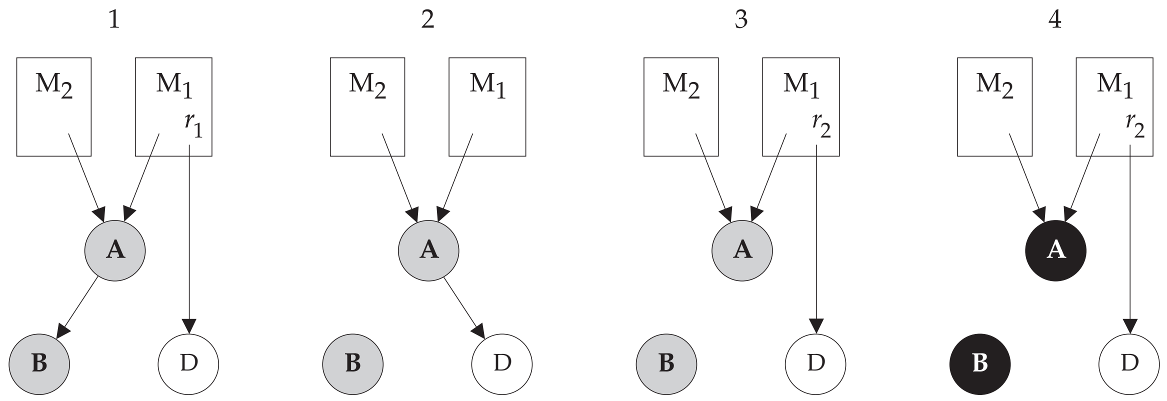


C	M		
$status_C \leftarrow Sync_1$			
	$MarkGray(A)$		
	$MarkGray(D)$	*Cooperate*	
	$status_M \leftarrow Sync_1$		1
	$r \leftarrow B$	**lecture**	2
	$A \leftarrow D$		
	$MarkGray(D)$	$Update(A, i, D)$	3
fin de *Mark*			
Scan : noircit A et D			4
Sweep : désalloue B, qui est accessible			

Figure 4.6: Si on ne marque pas l'ancienne valeur

Nous ne pouvons pas imposer au mutateur de griser B au moment de l'affectation $r \leftarrow B$ car cette affectation est une opération de **lecture**, et la contrainte 1 nous interdit d'imposer le moindre surcoût à cette opération. Pour résoudre ce problème, nous ferons en sorte que tous les objets qui sont accessibles au moment de la poignée de mains soient marqués par le cycle de GC qui a démarré avec cette poignée de mains. Pour ce faire, on oblige le mutateur à marquer en gris l'ancienne valeur (si elle était blanche) avant d'effectuer l'écriture.

L'algorithme de base fonctionne de la même façon si on marque l'ancienne valeur avant l'écriture ou si on marque la nouvelle après. Les auteurs de [33] préfèrent la deuxième solution car elle permet au GC de récupérer certains des objets devenus inaccessibles pendant le cycle courant. Avec la première solution, ces objets seront tous flottants. Cependant les calculs de Wadler [93] montrent que, même avec la deuxième solution, la plupart des objets rendus inaccessibles par **modification** seront flottants.



C	M_1	M_2	
$status_C \leftarrow Sync_1$	$(r_1 = D)$		
		$MarkGray(A)$, $status_{M_2} \leftarrow Sync_1$ — *Cooperate*	
		$MarkGray(B)$ — début de $Update(A, i, \mathbf{nil})$	1
	$MarkGray(B)$, $A \leftarrow D$ — $Update(A, i, D)$		
	$r_1 \leftarrow \mathbf{nil}$ **manipulation**		
	$MarkGray(A)$, $status_{M_1} \leftarrow Sync_1$ — *Cooperate*		2
	$r_2 \leftarrow D$ **lecture**		
		$A \leftarrow \mathbf{nil}$ — fin de $Update(A, i, \mathbf{nil})$	3
fin de *Mark*			
Scan : noircit A et B			4
Sweep : désalloue D, qui est accessible par M_1			

Figure 4.7: L'ancienne valeur peut changer pendant l'opération *Update*

4.3.2 Interférences entre les mutateurs

Nous définissons la *nouvelle valeur* comme la valeur qui est écrite par l'instruction d'écriture et l'*ancienne valeur* comme la valeur qui est remplacée par la nouvelle valeur lors de cette écriture.

Nous avons vu dans la section précédente que le mutateur doit marquer l'ancienne valeur avant d'effectuer une écriture. Supposons que l'on implémente l'opération *Update* par :

$$
\begin{aligned}
&Update(x, i, y) \equiv \\
&\quad MarkGray(heap[x, i]) \\
&\quad heap[x, i] \leftarrow y
\end{aligned}
$$

L'appel à *MarkGray* marque l'objet A pointé par le champ numéro i de l'objet x.

L'affectation $heap[x,i] \leftarrow y$ écrit y dans ce champ (en principe à la place du pointeur sur A). L'ancienne valeur est la valeur de $heap[x,i]$ juste avant cette écriture. Ce n'est pas forcément la valeur qui a été marquée par *MarkGray* car *Update* n'est pas atomique : la valeur de $heap[x,i]$ peut changer entre le moment où le mutateur exécute la première instruction de *Update* et le moment où il exécute la deuxième instruction.

Ce problème est illustré par la figure 4.7, dans laquelle M_1 tend un piège à M_2 en plaçant un pointeur vers un objet blanc (D) dans A entre les deux instructions de la procédure $Update(A, i, \mathbf{nil})$ exécutée par M_2. M_2 déclenche le piège en terminant son exécution de *Update* par l'affectation $A \leftarrow \mathbf{nil}$, qui efface un pointeur vers un objet blanc, empêchant ainsi le collecteur de repérer D comme objet vivant. D reste vivant car M_1 "jongle" avec le pointeur vers D : il le place dans A et l'efface de sa racine r_1 avant d'appeler *Cooperate*, puis il le relit dans sa racine r_2 avant que M_2 l'efface de A.

Le but de notre nouvelle procédure *Update* était justement d'empêcher l'écriture (effectuée par M_2) qui efface un pointeur vers un objet blanc. Cette nouvelle procédure n'est donc pas suffisante dans un système à plusieurs mutateurs.

La solution utilisée dans [37] est d'obliger le mutateur à marquer non seulement l'ancienne valeur avant l'écriture, mais aussi la nouvelle valeur après. Notre troisième version de *Update* est donc :

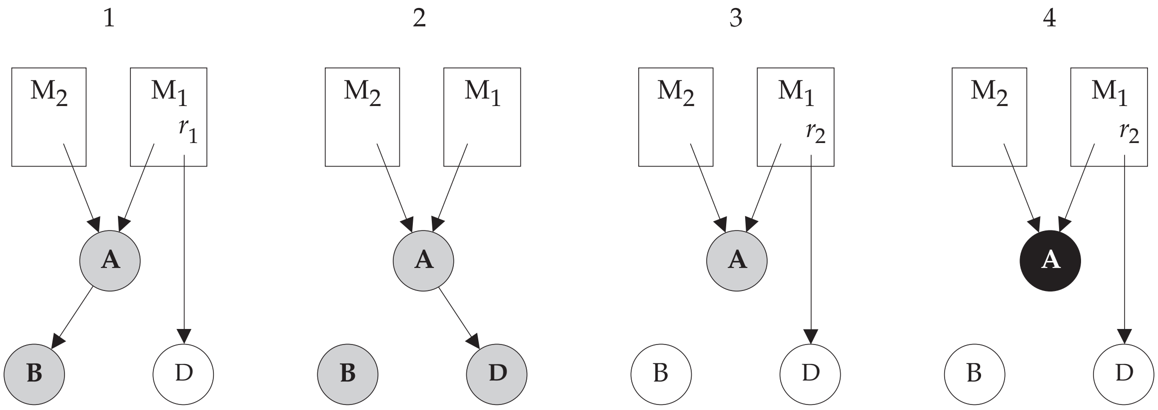
$Update(x, i, y) \equiv$
 $MarkGray(heap[x, i])$
 $heap[x, i] \leftarrow y$
 $MarkGray(y)$

Malheureusement, le GC peut effectuer un cycle complet et supprimer ce marquage, toujours entre deux instructions de M_2, comme on le voit sur la figure 4.8.

Ce scénario est presque le même que celui de la figure 4.7, mais M_1 appelle *Cooperate* et le collecteur effectue un cycle complet, toujours entre les deux instructions de $Update(A, i, \mathbf{nil})$ exécutées par M_2. Avec ce cycle, le collecteur annule l'effet du $MarkGray(D)$ effectué par le $Update(A, i, D)$ de M_1.

Pour empêcher ce scénario, nous ajoutons une nouvelle poignée de mains entre le collecteur et les mutateurs, qui empêche le collecteur d'effectuer un cycle complet entre deux instructions d'un mutateur. Nous plaçons cette nouvelle poignée de mains avant celle qui demande les racines : le collecteur commence par demander la première poignée de mains, à laquelle les mutateurs répondent simplement sans effectuer de travail supplémentaire, puis il demande les racines avec la deuxième poignée de mains. La seule contrainte est qu'un mutateur ne peut pas répondre à une poignée de mains s'il est en cours d'exécution de la procédure *Update*. On a donc une exclusion mutuelle entre *Update* et la réponse aux poignées de mains. Cette exclusion mutuelle s'implémente sans appel aux fonctions de synchronisation, par le simple fait que le mutateur est un processus séquentiel qui ne peut pas faire deux choses à la fois.

Pour gérer ces deux poignées de mains, nous utilisons deux constantes *Sync* et *Async*. La première poignée de mains commence lorsque $status_C$ prend la valeur *Sync* et se termine quand tous les mutateurs ont répondu en donnant à leur $status_m$ la valeur



C	M_1	M_2	
$status_C \leftarrow Sync_1$	$(r_1 = D)$		
		$\left.\begin{array}{l} MarkGray(A) \\ status_{M_2} \leftarrow Sync_1 \end{array}\right\} Cooperate$	
		$MarkGray(B)$	1
	$\left.\begin{array}{l} MarkGray(B) \\ A \leftarrow D \\ MarkGray(D) \end{array}\right\} Update(A, i, D)$		
	$r_1 \leftarrow \mathbf{nil}$ **manipulation**		
	$\left.\begin{array}{l} MarkGray(A) \\ status_{M_1} \leftarrow Sync_1 \end{array}\right\} Cooperate$		2
fin de *Mark* *Scan*, *Sweep* *Clear* : blanchit tous les objets *Mark* : $status_C \leftarrow Sync_2$		$Update(A, i, \mathbf{nil})$	
	$\left.\begin{array}{l} MarkGray(A) \\ status_{M_1} \leftarrow Sync_2 \end{array}\right\} Cooperate$		
	$r_2 \leftarrow D$ **lecture**		
		$A \leftarrow \mathbf{nil}$	3
		$\left.\begin{array}{l} MarkGray(A) \\ status_{M_2} \leftarrow Sync_2 \end{array}\right\} Cooperate$	
fin de *Mark* *Scan* : noircit A			4
Sweep : désalloue D, qui est accessible par M_1			

Figure 4.8: Si le GC n'attend pas avant de marquer

Sync. La deuxième poignée de mains se déroule de la même façon avec *Async*, mais les mutateurs doivent marquer leurs racines avant de répondre.

Nous avons donc une nouvelle version de l'étape *Mark* du collecteur :

Mark :
  $Status_C \leftarrow Sync$
  Attendre $(\forall m, status_m = Sync)$
  $Status_C \leftarrow Async$
  foreach $x \in Globals$ **do** $MarkGray(x)$
  Attendre $(\forall m, status_m = Async)$

La procédure *Cooperate* devient :

Cooperate $\equiv$
  if $status_m \neq status_C$ **then**
    if $status_C = Async$ **then** *Appeler MarkGray sur chaque racine.*
    $status_m \leftarrow status_C$

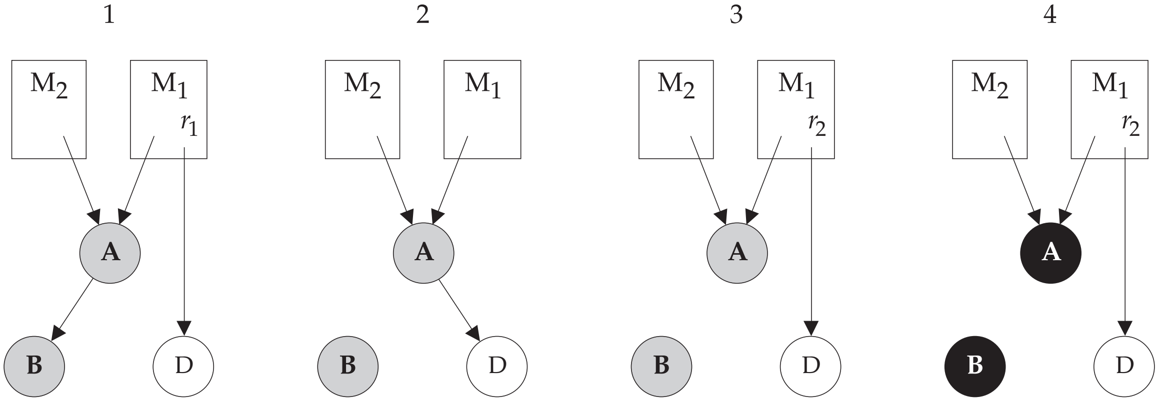
Cette nouvelle version de l'algorithme est peut-être correcte, mais elle est certainement inefficace. En effet, pendant les étapes *Sweep* et *Clear*, le collecteur n'examine que la couleur des objets, il ne consulte pas les pointeurs du graphe mémoire. Une **modification** du graphe mémoire pendant ces étapes ne peut donc pas interférer avec le travail du collecteur, et il suffirait d'effectuer directement l'écriture. Or notre troisième version de *Update* crée deux objets gris à chaque **modification**.

Il est particulièrement gênant de marquer des objets blancs en gris pendant l'étape *Clear*, car pendant cette étape les objets blancs accessibles sont ceux qui ont été blanchis en préparation du cycle suivant. Si un objet blanc A est rendu inaccessible par une **modification** pendant l'étape *Clear* du cycle n, il sera marqué en gris par *Update* et il restera gris jusqu'au début du cycle $n+1$. L'étape *Scan* du cycle $n+1$ va donc le noircir, et l'étape *Sweep* le considérera comme accessible. A sera blanchi par l'étape *Clear* du cycle $n+1$. Il ne sera pas marqué par les étapes *Mark* et *Scan* du cycle $n+2$, et il sera enfin désalloué par l'étape *Sweep* du cycle $n+2$.

L'objet A survit donc (avec tous ses descendants) à presque deux cycles complets du GC. Ce retard à la désallocation se traduira par une taille de tas plus importante et une dégradation des performances du programme.

Nous allons donc définir une quatrième version de *Update*, dans laquelle pc_C représente l'instruction courante du collecteur (celle qu'il est sur le point d'exécuter). Nous verrons plus tard comment le mutateur peut déterminer la valeur de pc_C par un test peu coûteux.

$Update(x, i, y) \equiv$
  if $(pc_C \in Mark \cup Scan)$ **then**
    $MarkGray(heap[x, i])$
    $MarkGray(y)$
  $heap[x, i] \leftarrow y$



C	M_1		M_2		
	$(r_1 = D)$				
	$pc_C \notin Mark \cup Scan$				
début de $Mark$					
$status_C \leftarrow Sync$					
		Update	$status_{M_2} \leftarrow Sync$	*Cooperate*	
			$pc_C \in Mark$		
			$MarkGray(B)$		1
	$A \leftarrow D$				
	$status_{M_1} \leftarrow Sync$	*Cooperate*			
	$r_1 \leftarrow$ **nil**	**manipulation**			2
$\forall m, status_m = Sync$				*Update*	
$status_C \leftarrow Async$					
	$MarkGray(A)$	*Cooperate*			
	$status_{M_1} \leftarrow Async$				
	$r_2 \leftarrow D$	**lecture**			
			$A \leftarrow$ **nil**		3
			$MarkGray(A)$	*Cooperate*	
			$status_{M_2} \leftarrow Async$		
$\forall m, status_m = Async :$ fin de $Mark$					
$Scan :$ noircit A et B					4
$Sweep :$ désalloue D, qui est accessible par M_1					

Figure 4.9: Il faut une troisième poignée de mains

Avec cette version, on a le scénario de la figure 4.9. Le problème est que M_1 teste la valeur de pc_C avant le début du cycle de GC, mais il n'utilise le résultat de son test qu'après que M_2 a appelé *Cooperate* et commencé un appel à $Update(A, i, \mathbf{nil})$. M_1 place alors un pointeur vers objet blanc (D) dans A, où M_2 va l'effacer, puis il effectue les deux poignées de mains et il récupère ce pointeur juste avant que M_2 l'efface en terminant son appel à *Update*.
Il y a trois solutions pour éviter ce scénario :

1. revenir à la troisième version de *Update*, qui marque toujours la nouvelle valeur,
2. ajouter un test dans *Update* pour marquer la nouvelle valeur si la première poignée de mains est en cours (sur la figure 4.9, M_1 devrait tester à nouveau la valeur de pc_C après son affectation $A \leftarrow D$),
3. ajouter une nouvelle poignée de mains avant les deux autres, pour obliger tous les mutateurs à remarquer que le collecteur est entré dans l'étape *Mark* avant de commencer la poignée de mains *Sync*.

Nous avons déjà rejeté la solution 1 car elle crée trop d'objets flottants. Nous préférons la solution 3 à la solution 2 car elle donne moins de travail aux mutateurs, comme nous allons le voir.

Dans la solution 3, nous utilisons trois valeurs pour les variables *status* : $Sync_1$, $Sync_2$ et *Async* et l'étape *Mark* du collecteur devient :

Mark :
$Status_C \leftarrow Sync_1$
Attendre $(\forall m, status_m = Sync_1)$
$Status_C \leftarrow Sync_2$
Attendre $(\forall m, status_m = Sync_2)$
$Status_C \leftarrow Async$
foreach $x \in Globals$ **do** $MarkGray(x)$
Attendre $(\forall m, status_m = Async)$

La procédure *Cooperate* ne change pas : la nouvelle poignée de mains est gérée par le même code que la deuxième puisqu'elle ne demande pas d'action de la part des mutateurs à part répondre. On voit donc que le coût de cette solution pour les mutateurs est très faible : une écriture par cycle de GC. Il est plus élevé pour le collecteur puisque celui-ci doit attendre que tous les mutateurs aient répondu. En pratique, il n'attend pas longtemps : les mutateurs répondent vite puisque cela leur demande très peu de travail.

4.3.3 Efficacité de la recherche des objets gris

L'algorithme de base a un problème d'efficacité évident : il parcourt le tas plusieurs fois pour chercher des objets gris pendant son étape *Scan*. Comme il produit de nouveaux objets gris pendant ce parcours, cette recherche peut devenir très inefficace. Au pire, le

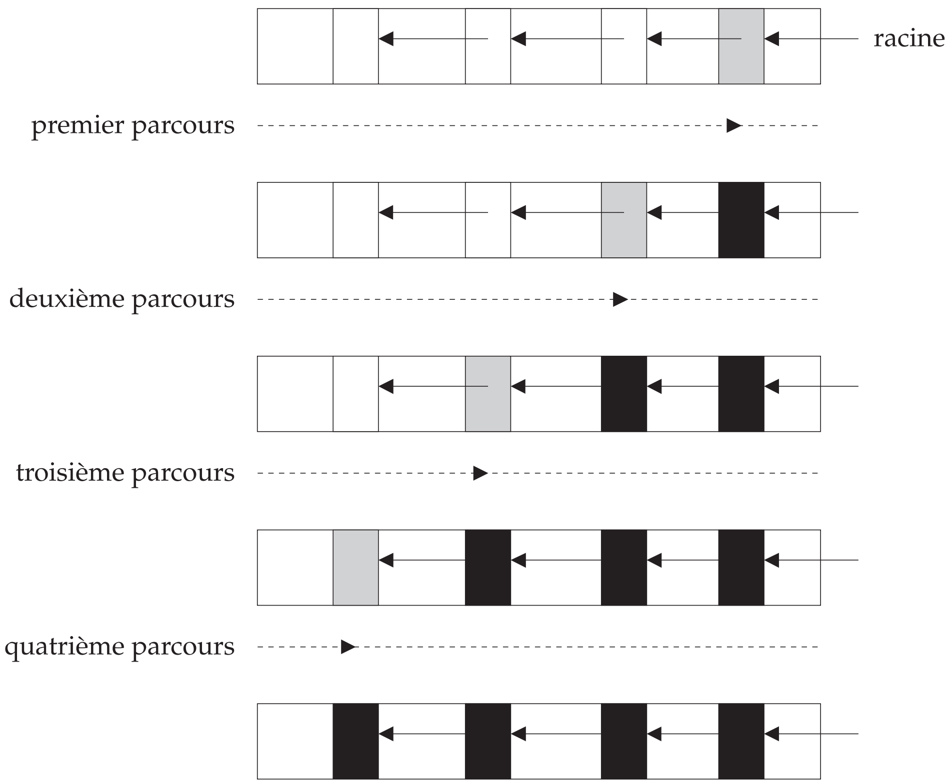


Figure 4.10: Parcours du tas pour trouver les objets gris

collecteur devra effectuer un parcours complet du tas pour trouver chaque objet gris, comme le montre la figure 4.10. Dans cette figure, le tas contient une liste dont les éléments sont placés dans l'ordre inverse du parcours du tas. A chaque fois qu'il trouve un objet gris, le collecteur le noircit et grise un autre objet dans la zone déjà parcourue. Pour trouver ce nouvel objet gris, il devra effectuer un nouveau parcours du tas.

Remarquons que la plupart des objets du tas qui sont gris ont été grisés par le collecteur lui-même, puisque les mutateurs ne grisent d'objets que lors des **modification**s, qui sont relativement rares. Nous allons donc modifier le collecteur pour mémoriser les objets qu'il grise dans une structure de données, que nous appellerons le *cache des objets gris*[2]. Lorsqu'il cherche un objet gris pour le noircir, le collecteur cherchera d'abord dans le cache. Il n'effectuera un parcours du tas que si le cache est vide.

Le GC ne doit pas utiliser une quantité arbitraire de mémoire. Nous utiliserons donc un cache de taille bornée (et très petite par rapport à celle du tas). Lorsqu'il n'y a plus de place dans le cache, le collecteur positionne *dirty* pour s'obliger à effectuer un nouveau parcours, pour chercher les objets gris qu'il n'a pas pu placer dans le cache.

On peut encore économiser le dernier parcours (celui qui commence quand il n'y a plus d'objet gris dans le tas). Cette économie est importante car un parcours du tas

[2]Il ne faut pas confondre ce cache, qui est une structure de données du programme, avec les caches de la machine, qui sont des circuits électroniques et qui ne sont pas contrôlés par le programme.

représente une fraction importante du coût total du GC. Pour réaliser cette économie, il suffit que le collecteur sache qu'il n'y a plus d'objet gris dans le tas. Pour obtenir cette information, nous transformons *dirty* en variable globale et nous obligeons les mutateurs à positionner *dirty* lorsqu'ils grisent un objet.

Le collecteur ne positionne plus *dirty* quand il rencontre un objet gris pendant son parcours, mais seulement s'il grise un objet déjà parcouru. La signification de la variable *dirty* change donc : dans l'algorithme de base, elle est vraie si le parcours courant a trouvé un objet gris ; dans notre algorithme, elle sera vraie si le parcours courant a raté un objet gris, donc si le parcours suivant trouvera un objet gris. Le collecteur s'arrête (comme dans l'algorithme de base) si un parcours se finit avec $dirty = \mathbf{false}$, puisqu'il sait alors qu'un parcours supplémentaire ne trouverait pas d'objet gris. Il évite donc d'effectuer ce parcours "à vide" (contrairement à l'algorithme de base).

Pour implémenter le cache, il faut choisir une structure de données qui permette d'effectuer très rapidement les deux opérations de base : ajouter un objet au cache et extraire un objet du cache. Le collecteur n'a pas besoin que l'opération d'extraction renvoie les objets dans un ordre précis : il a seulement besoin d'obtenir un objet gris quelconque lorsqu'il utilise cette opération. Les deux opérations (ajout et extraction) doivent être rapides car le collecteur va les exécuter très fréquemment.

Dans leur système, qui fonctionne avec un seul mutateur, Kung et Song [54] utilisent une queue à double entrée qui joue le même rôle que notre cache. Ils suppriment complètement les parcours du tas en s'assurant que tous les objets gris sont dans cette queue. Pour obtenir ce résultat, le mutateur doit placer dans la queue les objets qu'il grise. Il le fait à une extrémité de la queue, pendant que le collecteur utilise l'autre extrémité pour ajouter et extraire des objets gris de la queue. Il est possible d'implémenter cette structure de données sans utiliser de primitive de synchronisation, en reposant seulement sur l'atomicité des lectures et des écritures.

Cette queue n'est pas bornée, car elle doit contenir tous les objets gris (c'est-à-dire potentiellement tous les objets du tas). De plus, cette solution ne s'adapte pas à un système à plusieurs mutateurs : il faudrait une exclusion mutuelle pour protéger l'accès à l'extrémité de la queue utilisée par les mutateurs.

Contrairement à celui de Kung et Song, le fonctionnement de notre algorithme ne dépend pas du choix de la structure de données utilisée pour implémenter le cache, car celui-ci est local au collecteur. Nous prendrons donc une vue abstraite du cache : un multi-ensemble d'adresses.

var $cache \in$ **multiset of** $ADDR$

La variable *dirty* est maintenant une variable globale :

var $dirty \in BOOL$

Nous remplaçons la procédure *MarkGray* du mutateur par la procédure *MarkAndWarn*, qui positionne *dirty* lorsqu'un objet devient gris :

$MarkAndWarn(x) \equiv$
 if $color[x] = White$ **then**
  $color[x] \leftarrow Gray$
  $dirty \leftarrow$ **true**

L'étape *Scan* du collecteur devient, en ignorant pour le moment le problème du débordement du cache :

Scan :
 repeat
  $dirty \leftarrow$ **false**
  foreach $z \in ADDR$ **do**
   if $color[z] = Gray$ **then**
    placer z dans le cache
    while $cache \neq \emptyset$ **do**
     prendre un objet x dans le cache

$$Blacken(x) \begin{cases} \textit{noircir } x \\ \textit{griser les fils blancs de } x \textit{ et} \\ \qquad \textit{les placer dans le cache} \end{cases}$$

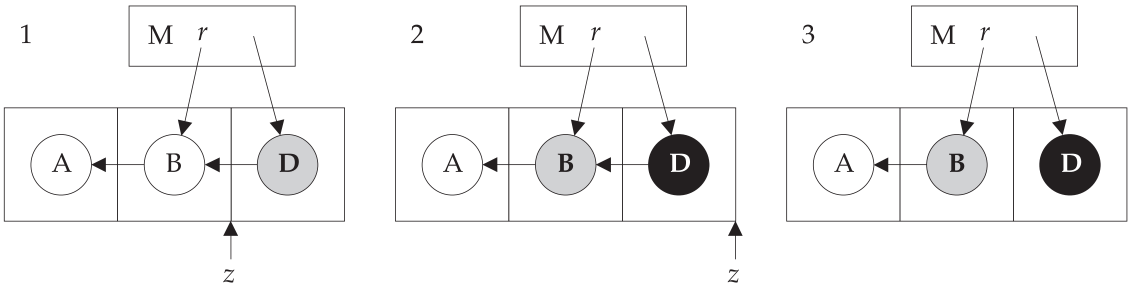
 until $\neg\, dirty$

Cette version ne fonctionne pas, comme le montre le premier scénario de la figure 4.11, dans lequel le tas contient trois objets, parcourus dans l'ordre A, B, D par la boucle **foreach** du collecteur. Au moment où le parcours du collecteur atteint D, avec $dirty$ = **false** et le cache vide, le mutateur change la couleur de B et s'apprête à positionner *dirty*. Le collecteur noircit alors D sans placer B dans le cache (puisqu'il est déjà gris), et il termine son étape *Scan* (puisque le cache est vide et le parcours terminé).

Pour remédier à ce problème, on peut essayer de positionner *dirty* avant de changer la couleur de B. Le deuxième scénario de la figure 4.11 montre que c'est parfaitement inutile : juste après que M a positionné *dirty*, le collecteur réinitialise *dirty* et recommence un parcours, ce qui le ramène à la même situation que dans le premier scénario.

Il faut donc que le collecteur ajoute au cache tous les fils gris de D quand il noircit D : non seulement ceux qu'il vient de griser mais aussi ceux qui étaient déjà gris. Il faut cependant éviter d'ajouter plusieurs fois au cache les objets partagés. On pourrait chercher dans le cache avant d'ajouter un objet, pour vérifier qu'il n'y est pas déjà. Nous avons retenu une solution plus efficace, qui consiste à noircir les objets au moment de les placer dans le cache. Ainsi, ils ne seront pas ajoutés une deuxième fois dans le cache car ils ne sont ni gris, ni blancs.

Nous avons donc une nouvelle version de l'étape *Scan* :



C	M			
	$r \leftarrow B$	**lecture**		
$dirty \leftarrow$ **false**				
$Scan$ avance jusqu'à D				1
	$color[B] \leftarrow Gray$			
$Blacken(D)$		$MarkAndWarn$		2
$\neg\, dirty$: fin de $Scan$			$Update$	
	$dirty \leftarrow$ **true**			
	$D \leftarrow$ **nil**			3
$Sweep$: désalloue A, qui est accessible par M				

C	M			
	$r \leftarrow B$	**lecture**		
	$dirty \leftarrow$ **true**			
$dirty \leftarrow$ **false**				
$Scan$ avance jusqu'à D				1
	$color[B] \leftarrow Gray$	$MarkAndWarn$	$Update$	
$Blacken(D)$				
$\neg\, dirty$: fin de $Scan$				2
	$dirty \leftarrow$ **true**			
	$D \leftarrow$ **nil**			3
$Sweep$: désalloue A, qui est accessible par M				

Figure 4.11: Le collecteur doit placer dans le cache les objets qui sont déjà gris

Scan :
repeat
$dirty \leftarrow$ **false**
foreach $z \in ADDR$ **do**
if $color[z] = Gray$ **then**
placer z dans le cache
$color[z] \leftarrow Black$
while $cache \neq \emptyset$ **do**
prendre un objet x dans le cache
$$Blacken(x) \begin{cases} \textit{noircir } x \\ \textit{noircir et placer dans le cache} \\ \qquad \textit{les fils blancs ou gris de } x \end{cases}$$
until $\neg\, dirty$

On remarque que cette version de *Scan* ressemble beaucoup au marquage récursif à deux couleurs décrit dans la section 2.3.2. Le collecteur n'utilise la couleur grise que pour communiquer avec les mutateurs. Comme on le verra en section 4.4.1, nous utiliserons aussi le gris pour gérer les cas de débordement du cache.

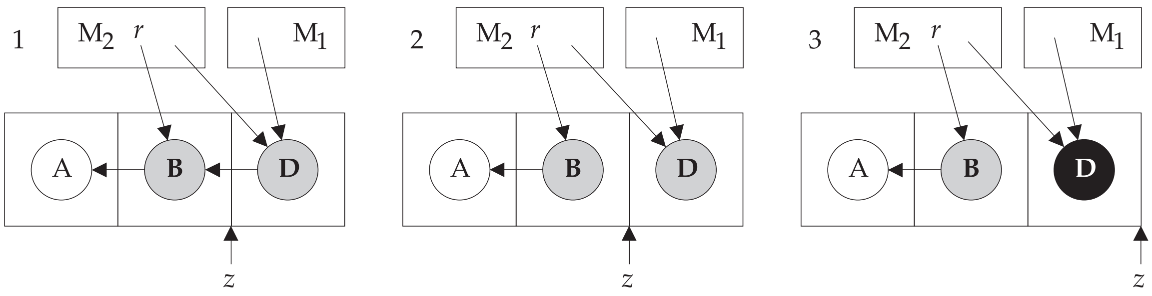
Il reste un problème avec cette nouvelle version, comme on le voit sur la figure 4.12. Le scénario est presque le même que le précédent, mais il y a maintenant deux mutateurs qui vont tous deux effectuer l'affectation $D \leftarrow$ **nil**. M_2 commence par $MarkAndWarn(B)$, qui change la couleur de B, et il s'apprête à positionner *dirty*. M_1 effectue alors sa **modification**, sans changer la couleur de B (car il est déjà gris) et donc sans positionner *dirty*. Le collecteur noircit alors D, qui n'a plus de fils, et il termine son étape *Scan*. M_2 positionne ensuite *dirty*, mais c'est trop tard.

Il faut donc que M_1 positionne *dirty* avant d'effectuer son écriture, même si ce n'est pas lui qui a changé la couleur de B.

Nous changeons donc *MarkAndWarn* pour positionner *dirty* même si x est déjà gris :

$MarkAndWarn(x) \equiv$
if $color[x] = White$ **then**
$color[x] \leftarrow Gray$
$dirty \leftarrow$ **true**
else if $color[x] = Gray$ **then**
$dirty \leftarrow$ **true**

Ce positionnement systématique de *dirty* signifie que le GC devra recommencer son parcours plus souvent. Pour éviter ce problème, nous transformerons en variable globale le pointeur z qui sert au collecteur pour son parcours de *Scan* et nous effectuerons le parcours dans l'ordre des adresses croissantes. Cela permet aux mutateurs de comparer la position de l'objet gris à celle de z avant de changer *dirty*. En effet, quand l'objet gris se trouve dans la partie du tas qui n'a pas encore été parcourue, il est inutile d'obliger le collecteur à effectuer un parcours supplémentaire, puisque le parcours qu'il est en train d'effectuer trouvera cet objet gris. Le code résultant de cette dernière modification est détaillé dans la section suivante.



C	M_1	M_2		
		$r \leftarrow B$	**lecture**	
$dirty \leftarrow$ **false**				
Scan avance jusqu'à D				
		$color[B] \leftarrow Gray$	*Update*	1
	$color[B] \neq White$ *Update*			
	$D \leftarrow$ **nil**			2
$Blacken(D)$				
$\neg\, dirty$: fin de *Scan*				
		$dirty \leftarrow$ **true**		
		$D \leftarrow$ **nil**		3
Sweep : désalloue A, qui est accessible par M_2				

Figure 4.12: Les mutateurs doivent positionner *dirty* même s'ils ne changent pas la couleur de l'ancienne valeur

4.4 Notre algorithme

Nous décrivons ici notre algorithme de GC concurrent à balayage. La section 4.4.1 décrit une version simplifiée, dans laquelle les tailles du tas et des objets sont fixes, et l'ensemble des mutateurs est constant. La section 4.4.2 explique comment prendre en compte un tas extensible, des objets de taille variable et les changements du nombre de mutateurs. Le modèle formel de cet algorithme est donné dans le chapitre 5.

4.4.1 Version simplifiée

Le tas et les variables globales

Le modèle du tas est le même que celui de l'algorithme de base :

const $Top, MaxIndex \in \mathbf{N}$
type $ADDR \equiv \{0, \ldots, Top-1\}$
$INDEX \equiv \{0, \ldots, MaxIndex\}$
var $heap \in$ **array** $[ADDR, INDEX]$ **of** $ADDR$

Nous ajoutons un ensemble fixe de mutateurs numérotés :

const $MaxPid \in \mathbf{N}$
type $PID \equiv \{0, \ldots, MaxPid\}$

Contrairement à l'algorithme de base, nous utilisons une vision abstraite de la liste libre. Elle est modélisée par un multi-ensemble (*alloc*), pour que nous puissions prouver qu'aucun objet ne s'y trouve en double. Notre liste libre ne fait pas partie du graphe mémoire, donc le graphe mémoire initial (l'ensemble des objets qui ne sont pas libres au moment où le programme commence) ne doit pas pointer vers la liste libre.

const $Globals \in$ **set of** $ADDR$
var $alloc \in$ **multiset of** $ADDR$
init $Globals \cap alloc = \emptyset$
$\forall x \in ADDR \setminus alloc, \forall i \in INDEX, heap[x,i] \in ADDR \setminus alloc$

Nous utilisons une quatrième couleur, le bleu, pour tous les objets du tas qui ne sont pas des nœuds du graphe mémoire. Ainsi l'étape de balayage du collecteur peut les ignorer facilement. Les objets bleus seront les objets de la liste libre globale et ceux des listes libres locales des mutateurs.
Initialement, tous les objets bleus sont dans la liste libre globale.

type $COLOR \equiv \{White, Gray, Black, Blue\}$
init $\forall x \in ADDR, color[x] = \begin{cases} Blue & \textbf{if } x \in alloc \\ White & \textbf{otherwise} \end{cases}$

Le cycle du collecteur est toujours divisé en quatre étapes : *Mark*, *Scan*, *Sweep* et *Clear*. Le cycle du GC commence par les trois poignées de mains, qui ont lieu pendant les étapes *Clear* et *Mark*. L'état initial du système sera donc la fin de l'étape *Clear*, juste avant la première poignée de mains.

Les poignées de mains sont implémentées comme dans la section 4.3.2 à l'aide de la variable $status_C$ et des variables $status_m$.

type $STATUS \equiv \{Async, Sync_1, Sync_2\}$
var $status_C = Async \in STATUS$
$\forall m \in PID$, **var** $status_m = Async \in STATUS$

Nous utilisons trois variables globales pour implémenter les améliorations décrites en sections 4.3.1 et 4.3.3 :

var $swept = +\infty \in ADDR \cup \{-\infty, +\infty\}$
$dirty \in BOOL$
$scanned = -\infty \in ADDR \cup \{-\infty\}$

La variable *swept* reflète la progression de l'étape *Sweep* ; elle vaut $-\infty$ avant et $+\infty$ après cette étape. Elle retombe de $+\infty$ à $-\infty$ au début de l'étape *Mark*. Lors des opération de **création**, le mutateur compare l'adresse de l'objet créé avec cette variable pour donner à l'objet la bonne couleur : blanc s'il a déjà été balayé, noir sinon, et gris si le mutateur ne sait pas (ce qui arrive si le collecteur est justement en train de balayer cet objet).

Le mutateur consulte aussi *swept* lors des opérations de **modification** pour savoir si le collecteur a terminé son étape *Scan* : si $swept \neq -\infty$ alors on n'est pas dans l'étape *Scan* et il est inutile de changer la couleur des objets et de positionner *dirty*.

La variable *dirty* sert à assurer que le collecteur ne termine pas son étape *Scan* avant d'avoir marqué tous les objets accessibles. Le collecteur l'initialise à **false** lorsqu'il démarre son parcours de recherche des objets gris. Si elle vaut **true** à la fin du parcours, le collecteur doit recommencer. Un mutateur la positionne à **true** lorsqu'il vient de marquer un objet en gris et qu'il détecte que le parcours risque de rater cet objet. La variable *scanned* reflète la progression du parcours. Le mutateur sait que le parcours ne ratera pas l'objet si son adresse est supérieure à *scanned*. Cette variable vaut $-\infty$ hors de l'étape *Scan*.

Les actions des mutateurs

Le GM ne prend en compte les calculs des mutateurs que dans la mesure où ils manipulent des pointeurs ou changent le contenu du tas. Nous fournissons donc un certain nombre de primitives aux mutateurs, qui peuvent les utiliser comme ils l'entendent à condition de respecter quelques contraintes simples (par exemple, ils doivent appeler *Cooperate* à intervalles assez rapprochés). Notre modèle des mutateurs est un ensemble de processus qui n'exécutent que ces actions, dans un ordre non déterministe.

Dans la suite de cette section, nous considérons le mutateur numéro m. Les variables locales du mutateur sont deux multi-ensembles de pointeurs dans le tas : ses racines (*roots*) et sa liste libre locale (*pool*). Les racines ne doivent pas pointer vers des objets de la liste libre globale.

$$
\begin{array}{ll}
\textbf{const} & MaxPool > 0 \in \mathbf{N} \\
\textbf{var} & pool \quad \in \textbf{multiset of } ADDR \\
 & roots \quad \in \textbf{multiset of } ADDR \\
\textbf{init} & pool = \emptyset \\
 & roots \cap alloc = \emptyset
\end{array}
$$

Nous définissons deux sous-programmes pour marquer les objets : *MarkGray* est utilisé quand le GC n'est pas dans l'étape *Scan* ; *MarkAndWarn* est utilisé pendant l'étape *Scan* pour s'assurer que le GC fera un parcours de plus si le parcours courant risque de rater l'objet marqué.

$$
\begin{array}{l}
MarkGray(x) \equiv \\
\quad \textbf{if } color[x] = White \textbf{ then } color[x] \leftarrow Gray \\
MarkAndWarn(x) \equiv \\
\quad \textbf{if } color[x] \neq Black \textbf{ then} \\
\quad\quad MarkGray(x) \\
\quad\quad \textbf{if } x \leq scanned \textbf{ then } dirty \leftarrow \textbf{true}
\end{array}
$$

Le mutateur doit exécuter l'action *Cooperate* à intervalles suffisamment rapprochés. Le coût de cette action est très faible, sauf dans le cas où il faut marquer les racines (quand $status_m$ passe de $Sync_2$ à $Async$), ce qui n'arrive qu'une fois par cycle.

Cooperate $\equiv$
  if $status_m \neq status_C$ **then**
    if $status_m = Sync_2$ **then foreach** $x \in roots$ **do** $MarkGray(x)$
    $status_m \leftarrow status_C$

La réservation de mémoire est la seule action qui nécessite une section critique. Celle-ci est notée par la construction "**await**...**do**...". La construction **pick** $x \in S$ choisit un élément du multi-ensemble S et le retire de S.

Reserve $\equiv$
  await $alloc \neq \emptyset$ **do**
    repeat
      pick $x \in alloc$ **do** $pool \leftarrow pool \oplus \{x\}$
    until $alloc = \emptyset \vee |pool| = MaxPool$

Les objets créés sont placés dans le graphe mémoire en changeant leur couleur de bleu à une autre couleur. L'action de création choisit la couleur du nouvel objet en fonction de l'état du GC. Si le balayage est en cours, les objets créés dans la partie du tas qui a déjà été balayée ne doivent pas être noirs. En effet, un objet noir le resterait jusqu'au cycle suivant et perturberait le marquage. De plus, les objets créés dans la partie du tas qui n'a pas encore été balayée ne doivent pas être blancs. Un objet blanc serait désalloué par le balayage en cours, même s'il est encore accessible. Dans les deux cas, on peut créer un objet gris : s'il a déjà été balayé, l'objet restera gris jusqu'au cycle suivant, mais un objet gris ne perturbe pas le marquage (il est traité comme un objet-racine supplémentaire), et dans l'autre cas, le balayage traite l'objet gris comme s'il était noir : il le blanchit et il ne le désalloue pas.

Si le GC est en cours de marquage, il faut créer l'objet en noir. En effet, le collecteur ne peut pas trouver de pointeurs vers cet objet (*a priori*, seule une racine pointe vers l'objet, et le collecteur a déjà marqué ses racines). L'objet ne doit donc pas être blanc. On peut le créer en gris, mais c'est donner du travail inutile au GC. En effet, les fils de cet objet sont forcément accessibles par ailleurs, donc ils seront marqués par le GC de toutes façons.

L'action de création pourrait donc utiliser le gris pour tous les objets créés, mais ce serait inefficace : tous les objets créés gris pendant le balayage survivent au moins à un cycle complet de GC, ce qui donne une quantité inacceptable d'objets flottants. L'action *Create* essaye donc de donner la bonne couleur à l'objet créé : noir ou blanc. Il n'est pas possible d'être certain de la bonne couleur, car l'ensemble des objets balayés peut changer pendant l'exécution de *Create*. Le cas ambigu est résolu par la création d'un objet gris. Pendant le marquage, on a $swept = -\infty$ et *Create* donne bien la couleur noire au nouveaux objets.

$Create \equiv$
  pick $x \in pool$ **do**
    $color[x] \leftarrow Black$
    if $status_m \neq Async \vee x < swept$ **then**
      $color[x] \leftarrow White$
    else if $x = swept$ **then**
      $color[x] \leftarrow Gray$
    return x

C'est la **modification** qui a le surcoût le plus élevé, mais notre décision d'utiliser trois poignées de mains nous permet de concentrer ce surcoût au début de l'action, avant l'écriture elle-même.

Entre le moment où il a répondu à la première poignée de mains et le moment où il a répondu à la troisième, le mutateur doit marquer l'ancienne et la nouvelle valeur, mais il n'a pas besoin de positionner *dirty* car l'étape *Scan* n'a pas encore commencé. Pendant l'étape *Scan* (qui est détectée en testant *swept*), le mutateur doit marquer l'ancienne valeur et positionner *dirty* si nécessaire :

$Update(x, i, y) \equiv$
  if $status_m \neq Async$ **then**
    $MarkGray(heap[x, i])$
    $MarkGray(y)$
  else if $swept = -\infty$ **then**
    $MarkAndWarn(heap[x, i])$
  $heap[x, i] \leftarrow y$

Remarquons que le test de pc_C utilisé en section 4.3.2 est remplacé ici par les tests $status_m \neq Async$ (qui indique que le collecteur est dans son étape *Mark*) et $swept = -\infty$ (qui indique l'étape *Scan*).

Enfin, il nous reste à parler des actions de **manipulation**, **lecture** et **remplissage**. Les deux premières n'ont pas besoin d'exécuter d'instructions spécifiques au GC : le mutateur est libre de dupliquer ou de supprimer une racine et de lire le contenu d'un objet du tas à tout instant, sauf bien sûr pendant qu'il exécute l'une des procédures définies dans cette section. Les actions de **manipulation** et de **lecture** se font donc directement par affectation. Quant au **remplissage**, il ne comporte pas non plus d'instructions supplémentaires (il se fait aussi par une simple affectation), mais il faut qu'un objet soit complètement rempli avant que le mutateur puisse l'utiliser. Il faut aussi que le mutateur remplisse tous ses objets avant de marquer ses racines (par *Cooperate*) : un objet non rempli a un contenu inconnu, et il ne faut pas que le collecteur le consulte pour essayer de trouver ses fils. En pratique, le mutateur remplira chaque objet juste après l'avoir créé.

Le collecteur

Le collecteur utilise la variable locale *phase* pour gérer les poignées de mains : la poignée de mains démarre lorsque le collecteur change $status_C$ et il donne à *phase* la valeur de

$status_C$ lorsque tous les mutateurs ont répondu (la poignée de mains est alors terminée). On a donc $phase \neq status_C$ si et seulement si une poignée de mains est en cours.

var $phase = Async \in STATUS$

Les valeurs possibles de *phase* sont :

$Sync_1$ après la première poignée de mains ; toute **modification** effectuée par un mutateur à partir de cet instant (et jusqu'à la fin de l'étape *Scan*) utilise une nouvelle valeur qui est grise ou noire.

$Sync_2$ après la deuxième poignée de mains ; toute **modification** effectuée par un mutateur à partir de cet instant (et jusqu'à la fin de l'étape *Scan*) remplace une ancienne valeur qui est grise ou noire.

Async après la troisième poignée de mains ; chaque mutateur a marqué ses objets-racines. L'étape *Mark* est donc terminée et le collecteur peut finir son cycle en effectuant les étapes *Scan* et *Sweep*.

La figure 4.13 illustre l'évolution des principales variables globales au cours d'un cycle de GC. Comme on peut le voir sur cette figure, l'étape *Clear* se termine avec le début de la deuxième poignée de mains, et l'étape *Mark* se termine juste après la fin de la troisième poignée de mains. En pratique, ces étapes sont de courte durée, et on a $phase = Async$ la plupart du temps.

Les mutateurs n'ont jamais besoin d'attendre lors des poignées de mains. C'est le collecteur qui doit attendre. Le collecteur utilise la procédure *Handshake* pour gérer les poignées de mains :

$Handshake(s) \equiv$
  $status_C \leftarrow s$
  foreach $m \in PID$ **do**
    await $status_m \neq phase$ **do skip**
  $phase \leftarrow s$

Le collecteur utilise un cache (*cache*). Il est de taille bornée : le travail du collecteur est de libérer de la mémoire, pas d'en consommer une quantité arbitraire.

const $MaxCache > 0 \in \mathbf{N}$
var $cache = \emptyset \in$ **multiset of** $ADDR$

Le marquage du graphe mémoire est fait par la procédure *Trace*. C'est une procédure récursive de parcours de graphe dont la pile de récursion est gérée explicitement : c'est le cache. La procédure *MarkBlack* noircit son argument et le place dans le cache s'il n'est pas déjà noir. La procédure *Trace* appelle d'abord *MarkBlack* pour placer son argument x dans le cache, puis elle entre dans une boucle qui répète l'instruction suivante : retirer un objet y du cache et mettre dans le cache tous les fils non noirs de y. La boucle s'arrête quand le cache est vide. Cet algorithme est exactement le parcours récursif du graphe décrit dans la section 2.3.2.

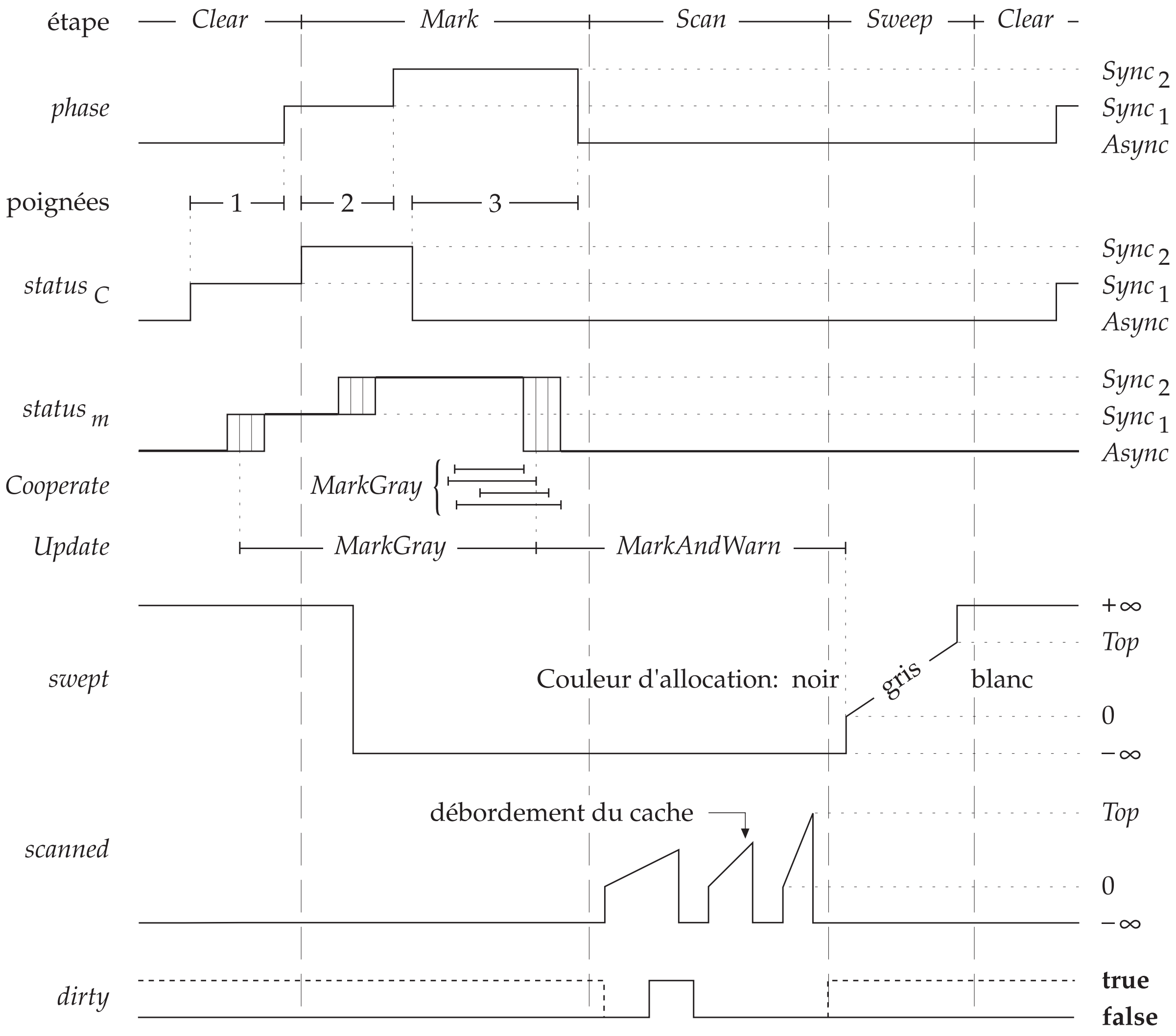


Figure 4.13: Évolution dans le temps des variables globales

Si le cache déborde, la procédure *MarkBlack* grise son argument au lieu de le noircir et de le placer dans le cache. Dans ce cas, lorsque *Trace* termine, elle n'a pas parcouru tout le graphe et les objets qu'elle a omis d'examiner sont gris. C'est le balayage effectué par l'étape *Scan* qui devra les retrouver.

$Trace(x) \equiv$
  $MarkBlack(x)$
  while $cache \neq \emptyset$ **do**
    pick $y \in cache$ **do**
      foreach $i \in INDEX$ **do** $MarkBlack(heap[y, i])$

$MarkBlack(x) \equiv$
  if $color[x] \neq Black$ **then**
    if $|cache| < MaxCache$ **then**
      $color[x] \leftarrow Black$
      $cache \leftarrow cache \oplus \{x\}$
    else
      $color[x] \leftarrow Gray$
      if $x < scanned$ **then** $dirty \leftarrow$ **true**

Le code du collecteur est donné en figure 4.14. L'étape *Mark* marque les objets pointés par les variables globales et demande aux mutateurs de marquer leurs objets-racines. Elle utilise la construction "**cobegin**...**and**..." pour exécuter les deux dernières poignées de mains en parallèle avec le marquage des variables globales.

L'étape *Scan* effectue un parcours séquentiel du tas pour trouver les objets gris, et un parcours récursif borné (avec la procédure *Trace*) à partir de chaque objet gris trouvé. Il est important de noter que la variable *scanned* pointe toujours vers l'objet dont le collecteur va tester la couleur, sauf entre le test $color[scanned] = Gray$ et l'incrémentation de *scanned*, où elle pointe vers l'objet précédent (celui qui vient d'être testé). Il est donc vrai à tout instant que tout objet dont l'adresse est strictement supérieure à *scanned* sera examiné par le parcours courant. C'est cette propriété qui justifie le test effectué par *MarkAndWarn* avant de positionner *dirty* : si l'adresse est inférieure ou égale, l'objet ne sera peut-être pas examiné et il faut positionner *dirty*.

L'étape *Sweep* effectue un balayage normal, qui blanchit les objets noirs et désalloue les objets blancs. Les objets désalloués sont d'abord coloriés en bleu puis placés dans la liste libre. La construction "**await true do**..." implémente l'exclusion mutuelle pour l'accès à la liste libre.

De même que pour *scanned*, *swept* pointe toujours sur l'objet qui va être examiné ou sur l'objet précédent. C'est ce qui justifie les deux tests effectués par *Create* : si l'objet x se trouve avant *swept*, on est sûr qu'il a déjà été examiné. S'il est égal à *swept*, le mutateur n'a pas de certitude et il doit marquer l'objet en gris. Si cet objet a déjà été balayé, il restera gris jusqu'au prochain cycle de GC ; s'il n'a pas encore été balayé, il sera traité comme s'il était noir par le code de balayage.

Enfin, l'étape *Clear* ne fait que la première poignée de mains car l'étape *Sweep* blanchit les objets accessibles en même temps qu'elle désalloue les objets inaccessibles.

4.4.2 Extensions

Dans cette section nous expliquons comment étendre l'algorithme décrit dans la section précédente pour le faire fonctionner avec un tas extensible et un nombre variable de mutateurs. Ces extensions sont absolument nécessaires à une implémentation réaliste, et il est indispensable de les intégrer au modèle formel de l'algorithme car elles ont des conséquences compliquées sur la preuve de correction. Nous ne donnons pas le pseudo-code de ces extensions, mais le modèle formel exposé au chapitre 5 les intègre.

```
Clear :
    Handshake(Sync_1)
Mark :
    swept ← −∞
    cobegin
        Handshake(Sync_2)
        Handshake(Async)
    and
        foreach x ∈ Globals do Trace(x)
Scan :
    repeat
        dirty ← false
        scanned ← 0
        while scanned < Top do
            if color[scanned] = Gray then Trace(scanned)
            scanned ← scanned+1
        scanned ← −∞
    until ¬ dirty
Sweep :
    swept ← 0
    while swept < Top do
        if color[swept] ∈ {Black, Gray} then
            color[swept] ← White
        else if color[swept] = White then
            color[swept] ← Blue
            await true do alloc ← alloc ⊕{swept}
        swept ← swept+1
    swept ← +∞
```

Figure 4.14: Le cycle du collecteur

Gestion des processus

En imposant un ensemble fixe de mutateurs, nous avons jusqu'à présent obligé le programme à se composer d'un nombre fixe de processus. Il est important de supprimer cette restriction pour obtenir un système plus souple, dans lequel les processus peuvent être créés et détruits. Un tel système fournit au programmeur une primitive **exit** pour détruire le processus qui l'exécute, et une primitive **launch** qui permet à un processus de démarrer un autre processus.

La primitive **exit** ne pose pas de problème particulier, si ce n'est qu'il faut prévenir le collecteur de ne plus attendre de réponse de ce processus lors des poignées de mains.

La primitive **launch** pose un problème de vivacité lors des poignées de mains. En effet, la procédure *Handshake* énumère l'ensemble des processus vivants pour attendre qu'ils aient tous répondu. Si on ajoute des processus à cet ensemble pendant la poignée de mains, et si les processus sont ajoutés avec un *status* qui indique qu'ils n'ont pas répondu, alors la poignée de mains risque de ne jamais se terminer.

Pour éviter ce problème, nous donnons à chaque nouveau processus le *status* de celui qui l'a lancé, et nous interdisons **launch** aux processus qui n'ont pas encore répondu à une poignée de mains en cours. Les mutateurs doivent donc appeler *Cooperate* avant **launch**, ce qui garantit que les nouveaux processus ne seront pas créés avec le "mauvais" *status*.

Il ne suffit pas que le nombre de processus qui ont le "mauvais" *status* soit fini. Il faut aussi que le nombre de tours dans la boucle de *Handshake* soit fini. Autrement dit, le collecteur doit pouvoir s'assurer que tous les processus ont répondu à la poignée de mains (qu'ils ont le "bon" *status*) sans tester chaque processus nouvellement créé, sinon une création continuelle de processus pourrait empêcher le collecteur de sortir de la procédure *Handshake* et donc bloquer la récupération de mémoire.

Pour résoudre ce problème, nous utilisons deux listes de processus protégées par un sémaphore. La première liste contient tous les processus qui n'ont pas encore répondu à la poignée de mains, et la deuxième tous les processus qui ont répondu. Les processus nouvellement créés sont insérés dans la deuxième liste (avec le "bon" status) et la procédure *Handshake* parcourt la première liste, qui ne peut pas s'agrandir. À la fin d'une poignée de mains et entre deux poignées de mains la première liste est vide. Au début d'une poignée de mains, le collecteur échange les deux listes.

Cette utilisation d'un sémaphore lors de la création de processus ne respecte pas notre contrainte 2. Cela n'a pas d'importance en pratique car la primitive **launch** est beaucoup plus coûteuse qu'une synchronisation. Si on tient absolument à respecter cette contrainte, Georges Gonthier [43] a inventé une structure de données qui permet d'implémenter la gestion des processus sans exclusion mutuelle.

Gestion du tas

Un gestionnaire de mémoire réaliste doit pouvoir traiter des objets de tailles différentes, étendre le tas, et recoller les objets désalloués adjacents pour lutter contre la fragmentation.

On parle d'objets de taille "variable". La taille d'un objet vivant ne varie pas au cours du temps mais les objets ne sont pas tous de la même taille, et l'allocation doit pouvoir couper un objet libre en deux pour allouer un nouvel objet plus petit. Ce découpage des objets interfère de façon complexe avec le balayage du collecteur, comme nous le verrons dans les chapitres 5 et 6 : si on utilise la partie basse de l'objet découpé pour en faire le nouvel objet, le balayage peut "oublier" des objets noirs alloués ultérieurement dans l'autre partie du bloc découpé.

Nous représentons donc le tas comme un ensemble de mots et non plus comme un ensemble d'objets. Chaque objet est représenté par un mot d'en-tête suivi par le contenu de l'objet (un nombre entier de mots contenant des pointeurs). Cette représentation nous oblige à balayer le tas dans l'ordre des adresses croissantes. En effet, à partir d'un objet A, on peut trouver l'objet suivant en ajoutant la taille de A à l'adresse de A. On ne peut pas trouver l'objet précédant A car on ne sait pas où se trouve son en-tête.

Cette convention pose un léger problème à cause de l'extension du tas, qui se fait par le haut (en augmentant *top*, la limite supérieure du tas). En effet, le balayage risque de ne jamais atteindre la fin du tas si celle-ci augmente trop vite. Pour éviter ce problème, nous définissons une variable *limit* qui est initialisée à la valeur de *top* avant le début de l'étape *Sweep*. Les objets créés au-delà de *limit* seront tous noirs, il est donc inutile de les balayer (aucun de ces objets ne sera désalloué dans ce cycle), mais il faut les blanchir avant le début du cycle suivant. Nous le faisons pendant l'étape *Clear*.

Chapitre 5

Modèle formel du GC concurrent

Ce chapitre présente le modèle formel détaillé de l'algorithme de GC concurrent décrit dans le chapitre 4. Le modèle formel est écrit en TLA, un formalisme inventé par Leslie Lamport [59, 22] qui permet d'écrire des programmes parallèles sous forme de formules de logique temporelle.

La section 5.1 décrit TLA et les notations que nous utilisons. La section 5.2 donne l'algorithme sous forme de formule TLA.

5.1 Le formalisme utilisé

5.1.1 TLA

TLA (*Temporal Logic of Actions*) est un formalisme logique proposé par Lamport pour spécifier et raisonner sur les systèmes concurrents (et en particulier les programmes parallèles). Un programme TLA est une formule de logique qui donne la relation de transition du programme, c'est-à-dire l'ensemble des états dans lesquels la machine pourra se trouver après l'exécution d'une étape du programme, pour chaque état de départ possible.

Nous ne donnons ici qu'une description succinte de TLA. En particulier, la partie temporelle ne sera pas décrite. Le lecteur est invité à se référer à [59, 1] pour une description plus complète de TLA.

Valeurs, variables et états

En TLA, toutes les informations qui ont une incidence sur le déroulement du programme sont contenues dans les variables. Soit *Var* l'ensemble de ces variables. Par exemple, le compteur de programme de chaque tâche (qui indique quelle est la prochaine instruction à exécuter) est une variable explicite en TLA.

Soit un ensemble *Val* de valeurs. Ce sont les données manipulées par le programme et par la preuve : booléens, entiers, chaînes de caractères, ensembles, multi-ensembles, etc.

La sémantique de TLA est définie en termes d'*états*. Un état s est une fonction $s : Var \to Val$. On appelle *signification* de x, et on note $[\![x]\!]$, la fonction $s \mapsto s(x)$. L'image de s par $[\![x]\!]$ est notée $s[\![x]\!]$. On appelle St l'ensemble des états.

Fonctions d'état et prédicats

Une *fonction d'état* est une expression construite à partir de variables et de valeurs. Par exemple $x+1$. La signification d'une fonction d'état f est une fonction $[\![f]\!] : St \to Val$. L'image de s par $[\![f]\!]$ (que l'on note $s[\![f]\!]$) est la formule f dans laquelle on remplace chaque variable v par $s[\![v]\!]$. Par exemple,

$$s[\![x+1]\!] = s[\![x]\!] + 1$$

La signification des fonctions d'état est donc un prolongement de la signification des variables. On appelle *prédicat* une fonction d'état à valeurs booléennes.

Actions

Soient s et t deux états, que nous appellerons respectivement l'état *courant* et l'état *suivant*. Pour construire des expressions dont certaines variables prennent leur valeur dans l'état courant et les autres dans l'état suivant, on utilise un opérateur *prime*, qui évalue son argument dans l'état suivant, le reste étant évalué dans l'état courant. On utilise cet opérateur pour construire les *actions* : une action est une expression booléenne formée de variables, de variables primées et de valeurs. Par exemple : $x'+1 = y$. On appelle la signification d'une action a la fonction $[\![a]\!] : St \times St \to Bool$ dont on obtient la valeur en (s,t) en remplaçant dans a chaque variable non primée v par $s[\![v]\!]$ et chaque variable primée w' par $t[\![w]\!]$. Par exemple, $s[\![x'+1 = y]\!]t$ est égal à $t[\![x]\!] + 1 = s[\![y]\!]$.

On note $s[\![a]\!]t$ l'image de (s,t) par $[\![a]\!]$. On dit qu'une action est *activée* dans l'état s s'il existe un état t tel que $s[\![a]\!]t$.

Programmes et comportements

Nous donnons ici une version très simplifiée de la partie temporelle de TLA, mais elle suffira pour comprendre la preuve de correction.

Notre programme est présenté comme une disjonction d'actions ; une telle disjonction est elle-même une action, donc notre programme est une action. On appelle *comportement* une suite d'états. Les programmes sont interprétés comme des prédicats sur les comportements : le programme p est vrai sur le comportement $(s_n)_{n\in\mathbf{Nat}}$ si $\forall n \in \mathbf{Nat}, s_n[\![p]\!]s_{n+1}$ est vrai.

5.1.2 Notations

Soit **Nat** l'ensemble des entiers naturels. Le symbole $\triangleq$ signifie “est égal par définition à”. Les multi-ensembles sont dénotés par le mot-clef **multiset**. Un multi-ensemble sur un ensemble S est une application de S dans **Nat**. L'ensemble des éléments d'un multi-ensemble M est noté $\widehat{M}$, c'est l'ensemble des éléments dont l'image par M est non nulle. L'union de multi-ensembles, notée $\oplus$, est la somme point à point des applications. La différence de multi-ensembles, notée $\ominus$, est la différence point à point des applications (avec $x - y = 0$ si $x \leq y$). Nous identifierons M et $\widehat{M}$ lorsque l'image de M est incluse dans $\{0, 1\}$. La différence de deux ensembles S et T est notée $S \backslash T$. Si R est une relation sur E, nous noterons $R(x)$ l'ensemble $\{y \in E | xRy\}$. Enfin, nous noterons $\bigcup P_E$ pour $\bigcup_{i \in E} P_i$.

Pour nos actions, nous utilisons une syntaxe plus proche d'Algol que de TLA. Cette syntaxe ne doit pas faire oublier que les actions sont des formules logiques plutôt que des instructions de programme. Voici un exemple qui illustre les principaux traits de la syntaxe utilisée :

$$
\begin{array}{l}
\textbf{type } A \\
\qquad S \triangleq \textbf{Nat} \\
\qquad C \triangleq \{B, W\} \\
\qquad H \triangleq \textbf{record} \begin{cases} c \in C \\ z \in S \end{cases} \\
\textbf{var} \quad r \in \textbf{multiset of } A \\
\qquad h \in \textbf{array } [A] \textbf{ of } H \\
\langle\ x \in r \\
\qquad\quad \Longrightarrow r \leftarrow r \ominus \{x\} \\
\qquad\qquad\quad h[x].c \leftarrow W\ \rangle
\end{array}
$$

Les déclarations de type définissent quatre ensembles : A que nous ne précisons pas ; S qui est égal à **Nat** ; C qui n'a que deux éléments, B et W ; H qui est le produit cartésien de C et S. La déclaration de H définit aussi les fonctions de projection c et z, que l'on applique avec la notation postfixe usuelle en informatique : si x est un élément de H, alors $x.c$ est la première composante de x.

En TLA, les variables ne sont pas typées. Nous ajoutons les types de variables pour aider à la compréhension du programme. La variable r contiendra un multi-ensemble d'éléments de A, et la variable h contiendra une fonction de A dans H.

Appelons a l'action de notre exemple. Elle est divisée en deux parties : une précondition et un corps, séparés par le symbole $\Longrightarrow$. La précondition est une formule logique, qui doit être vérifiée par l'état courant pour que a soit activée (si $s[\![a]\!]t$ est vrai alors s vérifie la précondition). Le corps est une conjonction d'affectations. Une affectation telle que $r \leftarrow r \ominus \{x\}$ (qui enlève x de r) est un prédicat sur les variables r', r et x (donc une action) que l'on écrirait $r' = r \ominus \{x\}$ en TLA orthodoxe.

L'affectation $h[x].c \leftarrow W$ illustre l'utilisation des enregistrements ; elle se traduit par $c(h'(x)) = W$.

Nous laissons implicite la quantification existentielle : chaque variable qui n'est pas une variable globale et qui apparaît libre dans une action est donc quantifiée en tête de cette action (par exemple x dans notre action a). De plus, toutes les variables qui ne sont pas mentionnées dans une action sont implicitement inchangées. Notre action

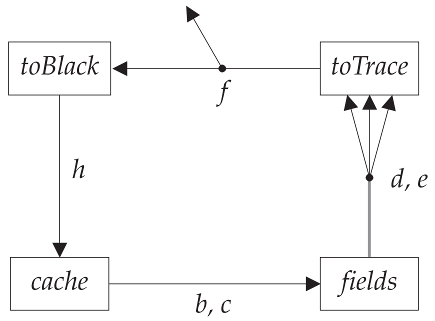


Figure 5.1: Circulation des données de la procédure *Trace*

a se traduit donc en TLA orthodoxe par

$$\begin{aligned}\exists x \in r : r' = r \ominus \{x\} & \\ & \wedge c(h'(x)) = W \\ & \wedge s(h'(x)) = s(h(x)) \\ & \wedge (\forall y \neq x, h'(y) = h(y)) \\ & \wedge \ \dots\end{aligned}$$

La partie omise contient les assertions de la forme $v' = v$ pour chaque variable v du programme sauf r et h.

Nous imposons une contrainte d'équité faible sur certaines actions en utilisant le symbole $\stackrel{w}{\Longrightarrow}$ à la place de $\Longrightarrow$. Cette contrainte interdit les comportement dans lesquels l'action est, à partir d'un certain instant, toujours possible et jamais exécutée. Autrement dit, l'action ne peut pas être indéfiniment activée sans être jamais exécutée.

Nous utilisons la notation **if** C **then**...**else**... pour dénoter la conjonction de deux implications dont les hypothèses sont respectivement C et $\neg C$.

5.2 L'algorithme

TLA favorise un style de programmation par “flot de données” plutôt que par une suite d'instructions impératives. Lorsque le pseudo-code du chapitre 4 utilise une variable x, la signification exacte de cette variable dépend de la valeur du compteur de programme pc. En TLA, nous utiliserons plusieurs ensembles pour représenter cette variable. Chaque étape du traitement est une action qui fait passer la valeur traitée d'un ensemble à un autre.

Prenons par exemple une version simplifiée des procédures *Trace* et *MarkBlack* du chapitre précédent :

$Trace(x) \equiv$
(a) $\quad MarkBlack(x)$
(b) $\quad$ **while** $cache \neq \emptyset$ **do**
(c) $\quad\quad$ **pick** $y \in cache$ **do**
(d) $\quad\quad\quad$ **foreach** $i \in INDEX$ **do**
(e) $\quad\quad\quad\quad MarkBlack(heap[y, i])$

$MarkBlack(x) \equiv$
(f) $\quad$ **if** $color[x] \neq Black$ **then**
(g) $\quad\quad color[x] \leftarrow Black$
(h) $\quad\quad cache \leftarrow cache \oplus \{x\}$

Les étapes du calcul de ces procédures seront représentées par quatre ensembles : *toTrace*, *toBlack*, *cache* et *fields* (figure 5.1). L'ensemble *toTrace* contient les valeurs que le collecteur a décidé de passer en argument à *MarkBlack*. Dans le pseudo-code, il est toujours vide sauf dans les états où le collecteur va exécuter la première instruction de *MarkBlack*, c'est-à-dire quand le compteur de programme du collecteur vaut (a), (e) ou (f). De *toTrace*, l'objet passe dans *toBlack* quand le collecteur teste sa couleur (exécute la ligne (f)), puis dans *cache* quand le collecteur noircit l'objet (quand il a exécuté (g) et (h)).

La boucle des lignes (d) et (e) est modélisée par l'ensemble *fields*, qui contient les indices des champs de l'objet y qui n'ont pas encore été examinés. Le corps de la boucle (la ligne (e)) retire un indice de *fields*, lit le pointeur qui correspond à cet indice, et place ce pointeur dans *toTrace* (en appelant *MarkBlack*).

Si on s'en tient au pseudo-code, les ensembles *toTrace* et *toBlack* ne peuvent contenir que zéro ou un élément, et les états dans lesquels ces ensembles ne sont pas vides correspondent à un point précis dans le pseudo-code ((f) et (g), respectivement). De même, l'ordre dans lequel on retire les éléments de *fields* est fixé par l'ordre de la boucle **foreach** et *fields* est vide sauf quand le collecteur est en train d'exécuter la boucle (quand il se trouve dans les lignes (d) et (e)).

Dans le modèle formel, nous utilisons une méthode plus générale : les données circulent dans le pipeline circulaire

$$\ldots \rightarrow toBlack \rightarrow cache \rightarrow fields \rightarrow toTrace \rightarrow toBlack \rightarrow \ldots$$

et il peut se trouver plusieurs objets en même temps dans *toBlack*. De plus, l'ordre dans lequel les objets passent de *fields* à *toTrace* n'est pas fixé. Le programme TLA est donc moins déterministe que le pseudo-code : l'ensemble des comportements qu'il décrit est plus grand, ce qui a deux avantages :

- L'implémenteur a une plus grande liberté de choix pour écrire un programme conforme au modèle. Par exemple, on peut changer la procédure *Trace* pour effectuer son parcours dans un ordre qui améliore la localité des références (c'est-à-dire qui groupe les accès à la mémoire pour améliorer les performances de la mémoire virtuelle).

- Paradoxalement, la preuve est facilitée par cette généralisation. En effet, il est plus facile de raisonner sur quatre ensembles en supprimant la variable "compteur de programme" que sur deux variables (x et i) et une demi-douzaine de valeurs du compteur de programme.

En TLA, les actions sont atomiques (il n'y a pas d'état intermédiaire entre deux états successifs d'un comportement). Nous nous assurons que nos actions peuvent s'implémenter sans utiliser de primitive de synchronisation en interdisant à une action d'accéder (en lecture ou en écriture) à plusieurs variables globales. Il y a quatre exceptions à cette règle :

- Les actions qui gèrent la liste libre. Nous avons vu qu'elles nécessitent une exclusion mutuelle.

- Les actions qui gèrent la liste des processus. On peut les implémenter avec une exclusion mutuelle ou avec une structure de données spéciale (section 4.4.2). Nous ne modélisons pas les détails de ces actions car ils n'interfèrent pas avec la preuve.

- Les actions qui lisent une variable globale que le processus courant (celui auquel appartient l'action) est le seul à pouvoir modifier. Ces actions nécessitent une exclusion mutuelle avec les autres actions du même processus. En implémentant ce processus par un programme séquentiel, on assure cette exclusion mutuelle sans faire appel à une primitive de synchronisation.

- Nous permettons aux mutateurs de lire la taille d'un objet accessible sans compter cette lecture comme un accès à une variable globale. De même, le collecteur peut lire la taille d'un objet blanc inaccessible (par les mutateurs). Ces entorses à la règle sont justifiées par le fait que la taille d'un objet ne change jamais tant qu'il est accessible, et seul le collecteur change la taille des objets blancs inaccessibles. On pourrait donc écrire un algorithme équivalent qui utiliserait à chaque fois une action séparée pour lire d'abord la taille de l'objet, et prouver que la taille ne peut pas changer entre cette action et la suivante. Nous ne l'avons pas fait car il faut que la taille de l'algorithme reste raisonnable.

Nous donnons maintenant le code de l'algorithme, avec quelques explications pour les variables et pour chaque action. Le code est aussi donné sous forme moins dispersée dans l'annexe A.

5.2.1 Déclarations globales

type *Pid*

$$
\begin{array}{ll}
\mathit{Addr} & \triangleq \mathbf{Nat} \\
\mathit{Sizes} & \triangleq \mathbf{Nat} \\
\mathit{Colors} & \triangleq \{\mathit{Blue}, \mathit{White}, \mathit{Gray}, \mathit{Black}\} \\
\mathit{Headers} & \triangleq \mathbf{record} \begin{cases} \mathit{color} \in \mathit{Colors} \\ \mathit{size} \;\; \in \mathit{Sizes} \end{cases}
\end{array}
$$

$$\begin{array}{ll} \mathit{Words} & \triangleq \mathit{Addr} \cup \mathit{Headers} \\ \mathit{Statuses} & \triangleq \{\mathit{Async}, \mathit{Sync}_1, \mathit{Sync}_2, \mathit{Dead}, \mathit{Avail}, \mathit{Quick}\} \end{array}$$

Nous avons un ensemble (*Pid*) d'identificateurs de processus. Les adresses (*Addr*) et les tailles d'objets (*Sizes*) sont des entiers naturels. Il y a quatre couleurs distinctes (*Colors*). Les en-têtes (*Headers*) sont des enregistrements à deux champs (couleur et taille). Un mot de la mémoire (*Words*) représente soit une adresse soit un en-tête. Les statuts possibles d'un mutateur (*Statuses*) sont les trois statuts correspondant aux poignées de mains, le statut *Dead* qui signale au collecteur un mutateur arrêté qu'il faut donc enlever de la liste des mutateurs actifs, et les statuts *Avail* et *Quick* qui correspondent aux mutateurs qui ne sont pas dans cette liste (disponibles et en cours de lancement, respectivement).

$$\begin{array}{lll} \textbf{var} & \mathit{heap} & \in \textbf{array } [\mathit{Addr}] \textbf{ of } \mathit{Words} \\ & \mathit{top} & \in \mathit{Addr} \\ & \mathit{dirty} & \in \mathit{Bool} \\ & \mathit{free}, \mathit{alloc} & \in \textbf{multiset of } \mathit{Addr} \\ & \mathit{status}_C & \in \mathit{Statuses} \\ & \mathit{swept} & \in \mathit{Addr} \uplus \{-\infty, +\infty\} \\ & \mathit{scanned} & \in \mathit{Addr} \uplus \{-\infty\} \end{array}$$

La mémoire (*heap*) est un tableau de mots indexés par les adresses. La variable *top* représente la limite courante du tas : le tas correspond aux adresses de l'intervalle semi-ouvert $[0, \mathit{top}[$. La variable globale booléenne *dirty* garde le même rôle que dans la section 4.4.1.

La liste libre est modélisée par deux multi-ensembles : *free* et *alloc*. Le premier contient l'ensemble des objets blancs qui ont été désalloués par le collecteur et le deuxième contient l'ensemble des objets bleus que les mutateurs peuvent réserver pour une allocation future. Le passage de *free* à *alloc* se fait par l'action qui colorie les objets en bleu. La variable *alloc* joue donc le même rôle que dans pseudo-code de la figure 4.14, et la variable *free* correspond à l'état transitoire du pseudo-code dans lequel le collecteur vient d'effectuer le test $\mathit{color}[\mathit{swept}] = \mathit{White}$: l'objet est (logiquement) désalloué, mais il n'est pas encore bleu.

Les variables status_C, *swept* et *scanned* jouent le rôle exposé dans la section 4.4.

$$\begin{array}{l} \forall m \in \mathit{Pid}, \\ \qquad \mathit{status}_m \in \mathit{Statuses} \\ \qquad \mathit{args}_m \ \ \in \textbf{multiset of } \mathit{Addr} \end{array}$$

Chaque mutateur m a une variable status_m pour les poignées de mains, et un multi-ensemble d'arguments args_m qui lui sert à hériter certaines racines du processus qui l'a lancé. Ce multi-ensemble est vide sauf pour les processus qui sont en cours de lancement.

Valeurs initiales

L'état initial du programme doit vérifier certaines conditions :

$$\begin{aligned}
\textbf{init}\ & top = 0 \\
& free = alloc = \emptyset \\
& status_C = Async \\
& swept = -\infty \\
& \forall m \in Pid, status_m \in \{Avail, Async\} \\
& |\{m \in Pid \mid status_m \neq Avail\}| < \infty \\
& \forall m \in Pid, args_m = \emptyset
\end{aligned}$$

Le tas et les listes libres sont vides, aucune poignée de mains n'est en cours, le balayage va démarrer, il y a un nombre fini de mutateurs et aucun mutateur n'est en cours de lancement.

5.2.2 Les actions de la liste libre

Cette section présente des actions agissant sur la liste libre qui sont indépendantes de l'algorithme de GC proprement dit. Dans une implémentation, elles peuvent être rattachées aux mutateurs, ou au collecteur, ou même être gérées par un processus indépendant.

Vue d'ensemble

Nous donnons ici une brève description informelle des actions. Les numéros entre parenthèses font références aux actions correspondantes.

Le système peut agrandir le tas en augmentant sa limite supérieure *top*. Il doit alors insérer dans la liste libre le bloc de mémoire ainsi ajouté au tas (1).

Lorsqu'on fait passer les objets de la liste *free* à la liste *alloc*, on doit simultanément changer leur couleur de blanc à bleu (2).

Le système peut retirer un objet blanc de la liste libre. Cela revient à allouer directement un objet inaccessible, que le GC devra récupérer. L'utilité de cette action est justifiée dans la description du code (3, 4).

Actions

$$\begin{aligned}
1\ \langle\ & s \in Sizes \\
& \Longrightarrow heap[top] \leftarrow \textbf{record}\begin{cases} color \mapsto Blue \\ size \ \ \mapsto s \end{cases} \\
& \qquad alloc \leftarrow alloc \oplus \{top\} \\
& \qquad top \leftarrow top + s + 1\ \rangle
\end{aligned}$$

Notre première action est l'action qui agrandit le tas. Elle met en place un en-tête d'objet bleu adjacent à la fin du tas, change la fin du tas pour coïncider avec la fin de cet objet, et place ce nouvel objet dans la liste libre. Cette action doit préserver le *pavage du tas* : le tas est entièrement composé d'objets disjoints et adjacents. Autrement dit, chaque mot du tas doit appartenir à un objet unique. Cette propriété est indispensable pour que le balayage fonctionne correctement : il ne peut pas y avoir de trous entre les objets.

Cette action préserve aussi le *recensement des objets bleus* : tout objet bleu doit appartenir à la liste libre ou à un mutateur vivant. Cette propriété est indispensable pour éviter les fuites de mémoire. En effet, le GC ne récupère pas les objets bleus. Si un objet bleu est "orphelin", il restera définitivement inaccessible.

$$
\begin{array}{l}
2 \ \langle \ x \in \mathit{free} \\
\qquad\qquad \Longrightarrow \mathit{heap}[x].\mathit{color} \leftarrow \mathit{Blue} \\
\qquad\qquad\quad\ \mathit{free} \leftarrow \mathit{free} \ominus \{x\} \\
\qquad\qquad\quad\ \mathit{alloc} \leftarrow \mathit{alloc} \oplus \{x\} \ \rangle
\end{array}
$$

La deuxième action fait passer un objet de *free* à *alloc* en le coloriant en bleu. Elle préserve aussi le recensement des objets bleus.

$$
\begin{array}{l}
3 \ \langle \ x \in \mathit{free} \\
\qquad\qquad \Longrightarrow \mathit{free} \leftarrow \mathit{free} \ominus \{x\} \ \rangle \\
4 \ \langle \ x \in \mathit{alloc} \\
\qquad\qquad \Longrightarrow \mathit{alloc} \leftarrow \mathit{alloc} \ominus \{x\} \\
\qquad\qquad\quad\ \mathit{heap}[x].\mathit{color} \leftarrow \mathit{White} \ \rangle
\end{array}
$$

Ces actions permettent d'allouer directement un objet blanc inaccessible, que le collecteur devra récupérer par la suite : la liste libre peut fuir comme un seau percé. On verra que ces actions sont indispensables au collecteur pour "voler" des objets de la liste libre en vue de regrouper plusieurs objets libres adjacents pour combattre la fragmentation. Les deux actions correspondent aux deux parties de la liste libre : l'action 3 retire un objet de *free* et l'action 4 de *alloc*.

5.2.3 Les actions d'un mutateur

Cette section présente les actions que le mutateur m peut exécuter. Il est libre de les exécuter dans un ordre quelconque, à condition bien sûr de n'exécuter que des actions activées et de respecter les contraintes d'équité faible. Par ailleurs, le mutateur est libre d'exécuter n'importe quelle action qui ne change aucune des variables de ce modèle (ce sont les *stuttering steps* de TLA). Les actions présentées ici représentent donc toutes les contraintes imposées par le GM aux mutateurs.

Variables locales du mutateur

Nous commençons par définir trois "types" qui sont des ensembles de valeurs que prendra le compteur de programme du mutateur. Ces valeurs correspondent donc à des étiquettes d'instructions dans un langage impératif.

$$
\begin{array}{lll}
\textbf{type} & \mathit{CreateProc} & \triangleq \{\mathit{Split}, \mathit{TestSweep}, \mathit{ClearNew}, \mathit{GrayNew}, \mathit{Fill}\} \\
 & \mathit{UpdateProc} & \triangleq \{\mathit{TestOld}, \mathit{GrayOld}, \mathit{TestScan}, \mathit{SetDirty}, \mathit{Store}\} \\
 & \mathit{Labels} & \triangleq \mathit{CreateProc} \uplus \mathit{UpdateProc} \uplus \{\mathit{Halt}, \mathit{Work}, \mathit{Launch}\}
\end{array}
$$

Les éléments de *CreateProc* sont les étapes de la procédure *Create*, les éléments de *UpdateProc* sont les étapes de *Store* et *Labels* est l'ensemble de toutes les valeurs possibles du compteur de programme : *Work* est l'état normal du mutateur, *Halt* représente l'état d'un mutateur à l'arrêt (après qu'il a appelé **exit**), et *Launch* représente un état transitoire d'un mutateur qui est en train de lancer un nouveau processus (c'est donc une étape de la procédure **launch**).

Les variables locales du mutateur m sont implicitement indicées par m dans cette section. Elles seront explicitement indicées dans la preuve.

var $pc = Halt \in Labels$
$roots = pool = toMark = toFill = \emptyset \in$ **multiset of** $Addr$
$answering = marking =$ **false** $\in Bool$
$child \in Pid$
$old, new, field \in Addr$

La variable pc est le compteur de programme du mutateur. Au début du programme, elle est égale à $Halt$, ce qui signifie :

- si $status_m = Avail$, que le mutateur est inactif,
- si $status_m = Async$, que le mutateur va démarrer.

Les variables *roots* et *pool* sont respectivement les racines et la liste libre locale de ce mutateur. La variable *toMark* correspond à la boucle de marquage des objets-racines : elle représente l'ensemble des objets qui restent à marquer par cette boucle. De même, *toFill* représente l'ensemble des champs que le mutateur doit encore remplir après avoir créé un nouvel objet.

La variable *answering* est vraie dans l'état transitoire de la procédure *Cooperate* où le mutateur vient de tester $status_m \neq status_C$ et n'a pas encore testé $status_m = Sync_2$. Dans cet état, le pseudo-code ne peut exécuter que ce deuxième test, mais notre modèle est moins restrictif : beaucoup d'actions peuvent s'exécuter même si *answering* est vraie.

De même, *marking* est vraie pendant l'exécution de la boucle de marquage des objets-racines. Lors du lancement d'un nouveau processus, la variable *child* contiendra le numéro de ce nouveau processus. Les variables *old*, *new* et *field* servent à la procédure *UpdateProc* (qui correspond à la procédure *Update* du chapitre 4). Elles contiennent respectivement l'ancienne valeur, la nouvelle valeur et l'adresse du mot de la mémoire qui va subir l'affectation. Les variables *old* et *new* servent aussi à la procédure *CreateProc*, pour stocker l'adresse du bloc libre utilisé et l'adresse du nouvel objet créé. Ce double emploi des variables est possible car le même mutateur n'exécute jamais en même temps les procédures *UpdateProc* et *CreateProc*.

Vue d'ensemble

Au commencement de son exécution, le mutateur effectue une action de démarrage, qui lui donne ses racines initiales et qui l'oblige à répondre si une poignée de mains est en cours (5).

Un mutateur peut lancer un nouveau mutateur. Il doit d'abord choisir un mutateur arrêté et le réserver (6). Il lui donne ensuite des arguments, qui serviront de racines initiales au nouveau mutateur (7). Enfin, il déclenche le démarrage du nouveau mutateur (8).

Un mutateur peut s'arrêter : il supprime toutes ses racines et se place dans un état de repos qui signale au collecteur que ce mutateur ne répondra plus aux poignées de mains (9).

La procédure *Cooperate* démarre lorsque le mutateur constate qu'une poignée de mains est en cours (10). Pour les deux premières poignées de mains, elle se contente de répondre ; pour la troisième, elle appelle la procédure de marquage des racines (11). La procédure de marquage s'assure que toutes les racines sont grises ou noires (13, 14). Lorsqu'elle se termine, le mutateur peut répondre à la troisième poignée de mains (12).

Le mutateur peut effectuer une manipulation locale des racines (15, 16) ou une lecture d'un objet accessible (17).

Le mutateur réserve de la mémoire en faisant passer des objets de la liste libre globale à sa liste libre locale (18, 19).

Pour créer un nouvel objet, le mutateur doit choisir un objet dans sa liste libre locale (20), lui donner provisoirement la couleur noire (20), puis lui donner la bonne couleur en fonction de sa position par rapport à *swept* (22, 23, 24). Si l'objet libre choisi est plus grand que l'objet à allouer, il faut d'abord le couper en deux (21).

Après avoir créé un objet, le mutateur doit l'initialiser en remplissant tous ses champs (25, 26).

Enfin, pour modifier le graphe mémoire, le mutateur choisit l'objet dans lequel il fera son affectation, lit l'ancienne valeur et choisit la nouvelle (27). Si le collecteur n'est pas dans son étape *Scan*, le mutateur peut alors effectuer son affectation (28, 33). Dans le cas contraire, il faut griser l'ancienne valeur (29, 30), positionner *dirty* si nécessaire (31, 32) et enfin effectuer l'affectation (33).

Démarrage du mutateur

$$
\begin{aligned}
5 \ \langle\ & pc = Halt \ \wedge\ status_m \in \{Async, Sync_1, Sync_2\} \\
& \overset{w}{\Longrightarrow}\ roots \leftarrow args_m \\
& \qquad args_m \leftarrow \emptyset \\
& \qquad answering \leftarrow (status_m \neq status_C) \\
& \qquad pc \leftarrow Work\ \rangle
\end{aligned}
$$

Le mutateur commence son existence avec $pc = Halt$ et $status_m$ désignant un statut actif. Le processus recopie ses arguments (qui ont été placés dans $args_m$ par le processus p qui a lancé m) dans ses racines. Le processus m a hérité du *status* de p. Si une poignée de mains est en cours, à laquelle p n'avait pas répondu, alors m doit immédiatement commencer à répondre, ce qu'il fait en positionnant *answering*, s'interdisant ainsi de lancer un nouveau processus avant d'avoir répondu, comme on le verra dans l'action suivante.

Cette condition est légèrement différente de celle que nous avons énoncé en section 4.4.2. Nous autorisons un mutateur à lancer un nombre fini de fils avant de répondre à une poignée de mains, mais nous interdisons à ces fils (qui ont le "mauvais" *status*) de lancer à leur tour de nouveaux mutateurs avant d'avoir répondu, pour éviter de créer une chaîne infinie de processus lancés avec le mauvais *status*, qui pourraient retarder indéfiniment la fin de la poignée de mains.

Notre action 5 donne enfin au *pc* du processus courant la valeur *Work*, qui indique l'état normal du processus, ce qui correspond dans le pseudo-code à l'état du mutateur quand il n'est pas en train d'exécuter l'une des procédures du GM.

Cette action porte une contrainte d'équité faible, qui est indispensable pour que le GC fonctionne correctement. En effet, un processus qui resterait bloqué avec $pc = Halt$ ne répondrait pas aux poignées de mains et il empêcherait donc le collecteur de fonctionner.

Lancement d'un nouveau mutateur

$$6 \ \langle\ pc = Work \wedge \neg\, answering \wedge p \in Pid \wedge status_p = Avail \\ \Longrightarrow child \leftarrow p \\ status_p \leftarrow Quick \\ pc \leftarrow Launch\ \rangle$$

Un mutateur au repos qui n'est pas en train de répondre à une poignée de mains peut décider de lancer un nouveau processus. Il choisit un numéro de processus p libre, et il le réserve par l'affectation $status_p \leftarrow Quick$. Il place p dans sa variable locale *child* pour communiquer la valeur de p entre les actions de lancement d'un nouveau mutateur.

Enfin, cette action marque l'entrée du mutateur m dans la procédure *Launch* et change la valeur de *pc* en conséquence. Cette valeur de *pc* désactive la plupart des actions du mutateur : les actions de *Launch* ne peuvent pas s'entrelacer avec, par exemple, les actions de *CreateProc* du même mutateur.

$$7 \ \langle\ pc = Launch \wedge x \in roots \wedge p = child \\ \Longrightarrow args_p \leftarrow args_p \oplus \{x\}\ \rangle$$

Cette action constitue la boucle principale de la procédure *Launch*. Le mutateur place un nombre quelconque de copies de ses racines dans les arguments de son fils.

$$8 \ \langle\ pc = Launch \wedge p = child \\ \overset{w}{\Longrightarrow} status_p \leftarrow \begin{cases} Async & \textbf{if } marking \\ status_m & \textbf{otherwise} \end{cases} \\ pc \leftarrow Work\ \rangle$$

Cette action marque la fin de la procédure *Launch* : le mutateur m donne à son fils son propre *status* et retourne à son état normal. Si m est en cours de marquage de ses propres racines (si *marking* est vrai), m donne d'avance à p un *status* égal à *Async*. En effet, m est en train de marquer ses racines, donc il est sûr qu'elles seront considérées comme vivantes par le collecteur. Or les racines de p seront ses arguments, qui constituent un sous-ensemble des racines de m. Il est donc inutile que p les marque aussi. On donne donc à p un statut indiquant qu'il a déjà marqué ses racines.

Cette action porte une contrainte d'équité faible : le mutateur ne doit pas rester indéfiniment dans *Launch* car cela bloquerait les poignées de mains et le collecteur ne pourrait plus travailler.

Suppression d'un mutateur

$$
\begin{array}{l}
9 \;\langle\; pc = \mathit{Work} \;\wedge\; \mathit{toMark} = \mathit{pool} = \emptyset \\
\qquad \Longrightarrow \mathit{status}_m \leftarrow \mathit{Dead} \\
\qquad\qquad \mathit{answering} \leftarrow \mathit{marking} \leftarrow \mathbf{false} \\
\qquad\qquad \mathit{roots} \leftarrow \emptyset \\
\qquad\qquad pc \leftarrow \mathit{Halt} \;\rangle
\end{array}
$$

Un mutateur au repos peut décider de disparaître, à condition que sa liste libre locale et l'ensemble des objets qu'il s'est promis de marquer soient tous les deux vides. Cette action rétablit les conditions initiales sur les variables locales du mutateur et affecte $\mathit{status}_m \leftarrow \mathit{Dead}$, ce qui signale au collecteur qu'il ne participera plus aux poignées de mains. Le collecteur se chargera de changer status_m en *Avail* : il désalloue non seulement les objets morts, mais aussi les processus morts.

Cette action ne vide pas explicitement *toFill* car l'implication

$$
pc = \mathit{Work} \;\Rightarrow\; \mathit{toFill} = \emptyset
$$

est un invariant de notre modèle, donc *toFill* est déjà vide lorsqu'on exécute cette action.

La procédure *Cooperate*

Les états de la procédure *Cooperate* sont représentés par les variables *answering* et *marking* plutôt que par des valeurs distinctes de *pc*. Cela signifie que les actions de *Cooperate* peuvent être entrelacées avec un certain nombre d'autres actions du mutateur. Les exceptions sont les procédures *Launch*, *CreateProc* et *UpdateProc*, qui ne peuvent pas démarrer si *answering* est vraie. Réciproquement, les actions 11 et 12, qui répondent aux poignées de mains, ne peuvent se déclencher que si $pc = \mathit{Work}$, ce qui empêche le mutateur de répondre aux poignées de mains si l'une des procédures *Launch*, *CreateProc* ou *UpdateProc* est en cours d'exécution.

$$
\begin{array}{l}
10 \;\langle\; pc \neq \mathit{Halt} \;\wedge\; \mathit{status}_m \neq \mathit{status}_C \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{answering} \leftarrow \mathbf{true} \;\rangle
\end{array}
$$

Cette action marque l'entrée dans la procédure *Cooperate* et le test qui constate qu'une poignée de mains est en cours. Le mutateur prend alors la décision de répondre à cette poignée de mains. La contrainte d'équité sur cette action matérialise l'obligation imposée au mutateur d'appeler *Cooperate* de temps en temps.

$$
\begin{array}{l}
11 \ \langle\ pc = Work \ \wedge\ answering \ \wedge \neg\, marking \\
\qquad \overset{w}{\Longrightarrow} answering \leftarrow \mathbf{false} \\
\qquad\quad \mathbf{if}\ status_m = Sync_2 \ \mathbf{then} \\
\qquad\qquad toMark \leftarrow toMark \oplus roots \\
\qquad\qquad marking \leftarrow \mathbf{true} \\
\qquad\quad \mathbf{else} \\
\qquad\qquad status_m \leftarrow \begin{cases} Sync_1 \ \mathbf{if}\ status_m = Async \\ Sync_2 \ \mathbf{if}\ status_m = Sync_1 \ \rangle \end{cases}
\end{array}
$$

Cette action est activée lorsque l'action 10 détecte qu'une poignée de mains a commencé. S'il s'agit de la première ou de la deuxième poignée de mains (si $status_m \neq Sync_2$), alors il suffit de sortir de la procédure *Cooperate* (par l'affectation $answering \leftarrow \mathbf{false}$) et de répondre à la poignée de mains en donnant la bonne valeur à $status_m$.

Si c'est la troisième poignée de mains, il faut marquer les racines avant de répondre. Cette action déclenche donc la procédure de marquage (en positionnant *marking*) et ajoute les racines de ce mutateur à l'ensemble *toMark* des objets à marquer. On ne change pas $status_m$ car il ne faut pas répondre avant d'avoir marqué les racines (donc avant la fin de la procédure de marquage).

Le rôle de la variable *answering* est de coder une contrainte d'équité forte assez complexe : si la condition $pc = Work \ \wedge\ status_m \neq status_C$ est vraie infiniment souvent, alors le mutateur doit exécuter infiniment souvent l'action 9 ou l'action 11. Pour ce faire, l'action 10 positionne *answering* dès que $status_m \neq status_C$, ce qui interdit à *pc* de changer avant que *answering* soit réinitialisé car les actions 6, 20 et 27, qui changent *pc*, sont inhibées par *answering*. La contrainte d'équité faible de l'action 11 est alors suffisante pour garantir qu'elle sera exécutée (à moins que l'action 9 ne le soit avant).

Enfin, dans le cas de la troisième poignée de mains, l'action 11 ne répond pas à la poignée de mains, mais elle réinitialise quand même *answering*. L'action 10 est donc réactivée et elle finira par repositionner *answering* (ce qui inhibe les procédures de création et de modification, et assure donc que l'action 12 finira par répondre à la poignée de mains). Ainsi, le mutateur peut exécuter dans l'intervalle les actions de création et de remplissage. Ces actions comportent un cas spécial si le marquage est en cours, pour placer dans *toMark* les objets nouvellement créés, en même temps qu'ils sont ajoutés aux racines.

$$
\begin{array}{l}
12 \ \langle\ pc = Work \ \wedge\ marking \ \wedge\ toMark = \emptyset \\
\qquad \overset{w}{\Longrightarrow} answering \leftarrow marking \leftarrow \mathbf{false} \\
\qquad\quad status_m \leftarrow Async \ \rangle
\end{array}
$$

Cette action marque la fin de la procédure *Cooperate* dans le cas où il fallait marquer les racines. Elle correspond à la fin de la procédure de marquage (qui est appelée par *Cooperate*). Cette fin est signalée par la condition $marking \ \wedge\ toMark = \emptyset$. La procédure *Cooperate* se termine en réinitialisant *answering* et *marking*, et en répondant à la poignée de mains. Cette action ne peut se déclencher que si $pc = Work$ car le mutateur ne doit pas répondre à la poignée de mains si une création ou une mutation est en cours.

Enfin, cette action porte une contrainte d'équité faible, qui interdit au mutateur de rester indéfiniment dans la procédure *Cooperate* : il ne peut pas laisser la poignée de mains en suspens, car cela bloquerait le collecteur.

La procédure de marquage

Les deux actions suivantes constituent la procédure de marquage, qui marque tous les objets de *toMark* (les objets que le mutateur s'est promis de marquer). Les objets de *toMark* qui sont déjà gris ou noirs n'ont pas besoin d'être marqués de nouveau.

$$13\ \langle\ x \in \mathit{toMark}\ \wedge\ \mathit{heap}[x].\mathit{color} \neq \mathit{White} \\ \quad\Longrightarrow \mathit{toMark} \leftarrow \mathit{toMark} \ominus \{x\}\ \rangle$$

Cette action retire de *toMark* un objet qui était déjà marqué (par exemple, par un autre mutateur).

$$14\ \langle\ x \in \mathit{toMark} \\ \quad\stackrel{w}{\Longrightarrow} \mathit{heap}[x].\mathit{color} \leftarrow \mathit{Gray} \\ \quad\quad \mathit{toMark} \leftarrow \mathit{toMark} \ominus \{x\}\ \rangle$$

Cette action retire un objet de *toMark* et le colorie en gris. Elle ne peut pas tester la couleur de l'objet avant de la changer, puisqu'il faudrait pour cela faire deux accès à la mémoire partagée (lecture et écriture) dans une action atomique. Il est donc possible pour un objet noir de redevenir gris par cette action. Comme le collecteur peut boucler si les objets noirs ne restent pas noirs, il faudra prouver que cette action ne peut griser qu'un nombre fini d'objets noirs à chaque cycle de GC.

Cette action porte une contrainte d'équité faible, donc elle ne peut pas rester indéfiniment activée sans se déclencher. Comme elle est activée dès que *toMark* est non vide, cela signifie que *toMark* doit finalement se vider, ce qui termine la procédure de marquage (en activant l'action 12).

Manipulation des racines

$$15\ \langle\ x \in \mathit{roots} \\ \quad\Longrightarrow \mathit{roots} \leftarrow \mathit{roots} \oplus \{x\}\ \rangle$$

Cette action représente la duplication d'une racine du mutateur.

$$16\ \langle\ x \in \mathit{roots} \\ \quad\Longrightarrow \mathit{roots} \leftarrow \mathit{roots} \ominus \{x\}\ \rangle$$

Cette action représente la suppression d'une racine du mutateur.

En général, une manipulation de racines telle qu'une affectation dans un registre correspond à la composition de ces deux actions : suppression de la racine correspondant à la valeur du registre avant l'affectation et duplication de la racine correspondant à la nouvelle valeur.

Lecture d'un objet du tas

17 $\langle\ x \in \mathit{roots}\ \wedge x < z \leq x + \mathit{heap}[x].\mathit{size}$
$\quad\Longrightarrow \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{heap}[z]\}\ \rangle$

Le mutateur peut à tout moment ajouter à ses racines un pointeur contenu dans un de ses objets-racines. En utilisant plusieurs fois cette action, le mutateur peut accéder à tout objet accessible. Remarquons que, si x est l'adresse d'un objet, alors $\mathit{heap}[x]$ est l'en-tête de cet objet, $\mathit{heap}[x].\mathit{size}$ est sa taille, et les pointeurs contenus dans cet objet sont stockés dans la mémoire aux adresses comprises entre x et $x + \mathit{heap}[x].\mathit{size}$.

Réservation de mémoire

18 $\langle\ \mathit{pc} = \mathit{Work}\ \wedge x \in \mathit{alloc}$
$\quad\Longrightarrow \mathit{alloc} \leftarrow \mathit{alloc} \ominus \{x\}$
$\quad\quad \mathit{pool} \leftarrow \mathit{pool} \oplus \{x\}\ \rangle$

Cette action permet au mutateur d'ajouter à sa liste libre locale un objet pris dans la liste libre globale. Quelle que soit la représentation de la liste libre globale, l'implémentation de cette action devra utiliser une primitive de synchronisation, car il faut au moins une lecture (pour choisir un x dans la liste libre) et une écriture (pour retirer x de la liste libre).

En pratique, une implémentation exécutera plusieurs fois de suite cette action dans une seule section critique. Cela reste conforme au modèle : l'implémentation n'aura que des comportements dans lesquels ces actions sont enchaînées, ce qui représente un sous-ensemble des comportements du modèle. Comme nous prouvons que tous les comportements du modèle sont corrects, ceux de l'implémentation le sont aussi.

19 $\langle\ x \in \mathit{pool}$
$\quad\Longrightarrow \mathit{pool} \leftarrow \mathit{pool} \ominus \{x\}$
$\quad\quad \mathit{alloc} \leftarrow \mathit{alloc} \oplus \{x\}\ \rangle$

Cette action permet au mutateur de rendre à la liste libre globale des objets qu'il avait précédemment réservés. Elle permet en particulier au mutateur de vider sa liste libre locale avant d'appeler **exit**. Elle permettra aussi au mutateur de rendre les objets libres qu'il ne peut pas utiliser (parce qu'ils sont plus petits que les objets qu'il alloue). C'est important pour que le GC puisse lutter contre la fragmentation en recollant les objets libres adjacents.

Création d'un objet

La création et le remplissage d'un objet représentent l'un des deux points délicats de notre algorithme (l'autre étant la procédure de modification du graphe mémoire). La création est nettement plus compliquée ici que dans le pseudo-code du chapitre 4 car nous prenons maintenant en compte les objets de taille variable.

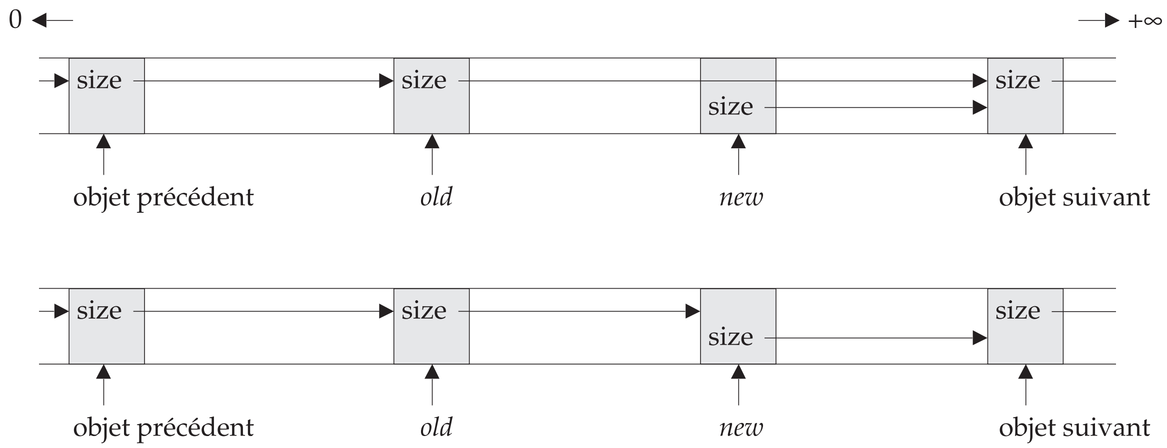


Figure 5.2: Découpage d'un objet libre

$$
20 \; \langle \; pc = Work \; \wedge \neg \, answering \; \wedge \; s \in Sizes \; \wedge \; x \in pool \; \wedge \; s \leq heap[x].\, size
$$

$$
\begin{aligned}
\Longrightarrow \; & pool \leftarrow pool \ominus \{x\} \\
& old \leftarrow x \\
& new \leftarrow x + heap[x].\, size - s \\
& toFill \leftarrow \{new + 1, \ldots, new + s\} \\
& heap[new] \leftarrow \mathbf{record} \begin{cases} color \mapsto Black \\ size \;\; \mapsto s \end{cases} \\
& pc \leftarrow \begin{cases} Split & \mathbf{if} \;\; old < new \\ TestSweep & \mathbf{otherwise} \end{cases} \rangle
\end{aligned}
$$

Cette action correspond au début de la procédure de création. Elle prend une taille s pour le nouvel objet et choisit un objet x de taille suffisante dans sa liste libre locale. Il y a alors deux cas possibles : si la taille de x est exactement s, alors la création se passe comme dans le pseudo-code (en particulier, on donne provisoirement au nouvel objet la couleur noire). Sinon, il faut couper en deux l'objet x : une partie deviendra l'objet créé, et l'autre partie retournera dans la liste libre.

On retire donc x de la liste libre locale et on le place dans la variable locale *old*, qui sert à communiquer avec les autres actions de la procédure de création. On calcule ensuite l'adresse *new* de l'objet créé. Si la taille de x est exactement s, on aura $new = old = x$, sinon *new* est "cadré" dans les adresses hautes de *old*. Le haut de la figure 5.2 illustre la situation après l'exécution de cette action. Les en-têtes sont représentés en gris, et la taille d'un objet est représentée par une flèche qui pointe sur l'objet suivant dans le chaînage du tas.

Après l'exécution de l'action 20 dans le cas où on découpe l'objet *old*, *new* n'est toujours pas un objet chaîné, même s'il a déjà un en-tête, car la taille de *old* n'a pas encore changé.

L'action 20 ajoute aussi à l'ensemble *toFill* les champs du nouvel objet. Cet ensemble est l'ensemble des champs qu'il faut remplir avant de pouvoir utiliser l'objet. Il est utilisé par la procédure de remplissage.

Enfin, si $new > old$, alors il faut finir le découpage de l'objet *old* : on active l'action 21 par l'affectation $pc \leftarrow Split$. Sinon on passe directement au test qui détermine la couleur à donner au nouvel objet : on active l'action 22 ($pc \leftarrow TestSweep$).

Toutes les actions suivantes de la procédure de création comportent une contrainte d'équité faible : lorsque cette procédure est commencée, elle doit se terminer en temps fini, notamment pour permettre aux poignées de mains d'avoir lieu.

$$
\begin{aligned}
21 \ \langle\ & pc = Split \\
& \overset{w}{\Longrightarrow} heap[old] \leftarrow \textbf{record} \begin{cases} color \mapsto Blue \\ size \ \ \mapsto new - old - 1 \end{cases} \\
& \quad pool \leftarrow pool \oplus \{old\} \\
& \quad pc \leftarrow TestSweep \ \rangle
\end{aligned}
$$

Cette action constitue la deuxième partie du découpage de l'objet *old*. L'en-tête du nouvel objet est déjà mis en place (par l'action 20) et il ne reste plus qu'à changer la taille de *old* pour que *new* devienne un objet chaîné, comme illustré par la figure 5.2.

Il est important d'effectuer les deux opérations dans cet ordre car le pavage du tas doit être préservé à tout instant : si on change d'abord la taille de *old* alors que *new* n'a pas encore un en-tête valide, le collecteur risque de lire l'en-tête invalide lors de son balayage.

Après avoir raccourci l'objet *old*, le mutateur le replace dans sa liste libre locale et il passe à l'action suivante pour continuer la procédure de création.

$$
\begin{aligned}
22 \ \langle\ & pc = TestSweep \\
& \overset{w}{\Longrightarrow} pc \leftarrow \begin{cases} ClearNew & \textbf{if } status_m \neq Async \lor new < swept \\ GrayNew & \textbf{if } status_m = Async \land old \leq swept \leq new \\ Fill & \textbf{otherwise} \ \rangle \end{cases}
\end{aligned}
$$

Cette action détermine la couleur à donner à l'objet nouvellement créé. Elle ne peut pas écrire directement la couleur dans l'en-tête de l'objet car elle effectue une lecture de la variable globale *swept*.

Si $status_m \neq Async$, le collecteur est dans son étape *Mark*. Avant le marquage des racines, on peut allouer l'objet en blanc, le marquage qui va commencer le trouvera s'il reste accessible jusque là. Pendant le marquage des racines, on l'alloue aussi en blanc car l'action 26 le placera dans *toMark* pour s'assurer qu'il sera marqué. Si $swept > new$, alors le collecteur est dans l'une des étapes *Sweep* ou *Clear* (comme le montre la figure 4.13), et l'objet *new* a déjà été balayé par le collecteur. Il faut donc l'allouer aussi en blanc (il ne sera plus examiné par le cycle courant, et il faut qu'il soit blanc pour le cycle suivant).

Si $status_m = Async$ et $swept < old$, alors le GC se trouve soit dans l'étape *Mark* ou *Scan* (et $swept = -\infty$) soit dans l'étape *Sweep* (et l'objet n'a pas encore été balayé par le GC). Dans les deux cas, il faut laisser l'objet noir. L'objet a donc sa couleur définitive, et on peut passer à la dernière phase de la création (*Fill*).

Enfin, si $status_m = Async$ et *swept* se trouve entre *old* et *new*, le mutateur ne sait pas si le collecteur va balayer l'objet ou non. En effet, si $swept = old = new$, le GC a examiné ou va examiner l'objet en cours de création. Le collecteur peut avoir examiné l'objet lorsqu'il était bleu, avant que l'action 21 le colorie en noir. Dans ce cas, il faut colorier l'objet en blanc ou en gris, sinon l'objet sera encore noir après le balayage. Le collecteur peut aussi examiner l'objet après l'exécution de l'action 22 et le changement de couleur. Dans ce cas, il faut colorier l'objet en gris ou en noir, sinon il sera désalloué par le GC avant même la fin de son allocation. Le mutateur ne peut pas distinguer ces deux cas, donc il doit colorier l'objet en gris pour être correct quoi qu'il arrive.

Si $swept = old \neq new$, le GC a examiné ou va examiner l'objet (bleu) que l'on a découpé. S'il l'a déjà examiné, il peut avoir lu son ancienne taille, et dans ce cas, il ne balayera pas l'objet *new*. S'il ne l'a pas examiné, il balayera normalement l'objet *new*. Le mutateur ne peut donc pas savoir si l'objet créé sera balayé par le collecteur, et il doit donc le colorier en gris. On change donc *pc* en *GrayNew*, ce qui active l'action qui coloriera *new* en gris.

Si $old < swept \leq new$, il est possible que *swept* ne pointe pas vers un en-tête d'objet valide si le GC vient de recoller des objets (par l'action 43). Dans ce cas, le GC a déjà calculé l'adresse de l'objet suivant, et celui-ci se trouve après *new*, donc le GC ne balayera pas l'objet *new*. Il se peut aussi que le GC ait lu l'objet *old* après son découpage par l'action 21. Dans ce cas, le GC balayera l'objet *new*. Dans le doute, le mutateur doit donc colorier l'objet *new* en gris.

$$
\begin{array}{l}
23\ \langle\ pc = \mathit{ClearNew} \\
\qquad \stackrel{w}{\Longrightarrow}\ \mathit{heap}[\mathit{new}].\,\mathit{color} \leftarrow \mathit{White} \\
\qquad\quad\ pc \leftarrow \mathit{Fill}\ \rangle
\end{array}
$$

Cette action blanchit l'objet *new*, en accord avec la décision prise par l'action 22. C'est la couleur définitive de l'objet, donc on passe à la dernière phase de *CreateProc*.

$$
\begin{array}{l}
24\ \langle\ pc = \mathit{GrayNew} \\
\qquad \stackrel{w}{\Longrightarrow}\ \mathit{heap}[\mathit{new}].\,\mathit{color} \leftarrow \mathit{Gray} \\
\qquad\quad\ pc \leftarrow \mathit{Fill}\ \rangle
\end{array}
$$

Cette action grise l'objet *new*, en accord avec la décision prise par l'action 22. On passe ensuite à la dernière phase de *CreateProc*.

Remplissage de l'objet créé

La procédure de remplissage des objets créés est représentée par les deux actions suivantes. L'action 25, qui effectue le remplissage proprement dit, ne teste pas la valeur de *pc*. Elle est donc activée dès que *toFill* contient des objets. Donc le remplissage est déclenché par l'action 20 et peut s'exécuter en même temps que les actions 21 à 24. La fin du remplissage (action 26), qui marque aussi la fin de la procédure de création, doit attendre que l'objet ait reçu sa couleur définitive.

$$
\begin{array}{l}
25\ \langle\ y \in \mathit{roots} \cup \{\mathit{new}\}\ \wedge\ z \in \mathit{toFill} \\
\qquad \stackrel{w}{\Longrightarrow}\ \mathit{heap}[z] \leftarrow y \\
\qquad\quad\ \textbf{if}\ \mathit{marking}\ \textbf{then}\ \mathit{toMark} \leftarrow \mathit{toMark} \ominus \{y\} \\
\qquad\quad\ \mathit{toFill} \leftarrow \mathit{toFill} \ominus \{z\}\ \rangle
\end{array}
$$

Cette action constitue la boucle principale du remplissage. L'ensemble *toFill* contient les champs qui restent à remplir. Le remplissage d'un champ consiste à y écrire une copie d'une racine y du mutateur (ou de l'objet *new* lui-même, qui est sur le point de devenir une racine). Si la procédure de marquage est active pendant que l'on remplit un objet, cet objet sera ajouté (par l'action 26) à l'ensemble *toMark*, donc on est sûr qu'il sera vu par le collecteur, donc on n'a pas besoin de marquer ses fils.

$$
\begin{array}{l}
26 \ \langle \ \mathit{pc} = \mathit{Fill} \ \wedge \ \mathit{toFill} = \emptyset \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{new}\} \\
\qquad\quad \textbf{if } \mathit{marking} \textbf{ then } \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{new}\} \\
\qquad\quad \mathit{pc} \leftarrow \mathit{Work} \ \rangle
\end{array}
$$

Cette action termine la procédure de remplissage. Quand l'ensemble *toFill* des champs à remplir est vide, tous les champs de l'objet *new* ont été remplis, et le mutateur peut enfin l'ajouter à ses racines. Si le marquage est en cours, il faut aussi mettre l'objet dans l'ensemble *toMark* des objets à marquer, car il devient accessible et il est encore blanc. Il faut donc qu'il subisse le même traitement que les objets qui étaient déjà alloués au moment où le marquage a commencé.

La procédure de modification du graphe mémoire

Les 7 actions suivantes constituent la procédure de modification. L'écriture qui change le graphe mémoire est la dernière action de cette procédure. Les autres actions établissent les conditions (notamment de couleurs des objets) nécessaires pour que cette écriture ne perturbe pas le travail du GC.

$$
\begin{array}{l}
27 \ \langle \ \mathit{pc} = \mathit{Work} \ \wedge \neg\, \mathit{answering} \ \wedge \ x, y \in \mathit{roots} \ \wedge \ x < z \leq x + \mathit{heap}[x].\mathit{size} \\
\qquad \Longrightarrow \mathit{new} \leftarrow y \\
\qquad\quad \mathit{field} \leftarrow z \\
\qquad\quad \mathit{old} \leftarrow \mathit{heap}[z] \\
\qquad\quad \textbf{if } \mathit{status}_m \neq \mathit{Async} \ \wedge \neg\, \mathit{marking} \textbf{ then } \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{new}\} \\
\qquad\quad \textbf{if } \mathit{status}_m = \mathit{Sync}_2 \textbf{ then } \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{old}\} \\
\qquad\quad \mathit{pc} \leftarrow \begin{cases} \mathit{TestOld} & \textbf{if } \mathit{status}_m = \mathit{Async} \\ \mathit{Store} & \textbf{otherwise} \ \rangle \end{cases}
\end{array}
$$

Le mutateur choisit deux objets-racines x et y, et l'adresse z d'un champ de x. L'affectation va remplacer le contenu de z par un pointeur sur y.

Cette action commence par placer y et z dans des variables locales du mutateur, pour que les autres actions puissent connaître les objets sur lesquels on travaille. Elle lit ensuite la valeur courante du contenu de z, qui représente *a priori* l'ancienne valeur (celle qui sera effacée par l'écriture).

Comme nous l'avons vu à la section 4.4.1, le mutateur doit s'assurer, au moins à partir de la fin de la première poignée de mains, que la nouvelle valeur sera marquée. Si status_m vaut Sync_1 ou Sync_2, il suffit de l'ajouter à *toMark*, et la procédure de marquage se chargera de la colorier en gris. Si $\mathit{status}_m = \mathit{Async}$ ou si la procédure de marquage est déjà commencée, le mutateur sait que *new* sera finalement marqué, car il est accessible et tous les objets accessibles par ce mutateur au moment où il commence son marquage devront être marqués par ce cycle de GC.

De même, entre la fin de la deuxième poignée de mains et la fin de l'étape *Scan*, il faut marquer l'ancienne valeur avant d'effectuer l'écriture. La deuxième poignée de mains se termine après que le mutateur a répondu, donc lorsque $status_m = Sync_2$. Lorsqu'il a répondu à la troisième poignée de mains, le mutateur doit non seulement marquer l'ancienne valeur mais aussi positionner *dirty*. Ce sont les actions 29 à 32 qui le font.

On passe donc à l'action 29 ($pc \leftarrow TestOld$) s'il faut colorier l'ancienne valeur en positionnant *dirty*, et on passe directement à l'écriture sinon.

$$
\begin{array}{l}
28 \ \langle\ pc \in UpdateProc \ \wedge\ swept > -\infty \\
\qquad\quad \Longrightarrow pc \leftarrow Store\ \rangle
\end{array}
$$

À tout moment de la procédure *UpdateProc*, le mutateur peut passer directement à l'écriture s'il se rend compte que le collecteur a fini l'étape *Scan* et commencé l'étape *Sweep*.

$$
\begin{array}{l}
29 \ \langle\ pc = TestOld \\
\qquad\quad \overset{w}{\Longrightarrow} pc \leftarrow \begin{cases} GrayOld & \textbf{if} \ \ heap[old].\,color = White \\ TestScan & \textbf{if} \ \ heap[old].\,color = Gray \\ Store & \textbf{otherwise} \ \ \rangle \end{cases}
\end{array}
$$

Cette action teste la couleur de l'objet *old*. S'il est blanc, il faut le griser et éventuellement positionner *dirty*. S'il est gris, le mutateur doit aussi positionner *dirty*, comme s'il avait grisé *old* lui-même.

$$
\begin{array}{l}
30 \ \langle\ pc = GrayOld \\
\qquad\quad \overset{w}{\Longrightarrow} heap[old].\,color \leftarrow Gray \\
\qquad\qquad\quad pc \leftarrow TestScan\ \rangle
\end{array}
$$

Cette action effectue le grisage de l'objet *old*, lorsque l'action 29 a décidé qu'il fallait le griser. Nous ne pouvons pas griser l'objet directement dans l'action 29 car il faudrait pour cela lire et écrire sa couleur dans la même action atomique. Après avoir grisé l'objet, on passe à l'action 31.

$$
\begin{array}{l}
31 \ \langle\ pc = TestScan \\
\qquad\quad \overset{w}{\Longrightarrow} pc \leftarrow \begin{cases} SetDirty & \textbf{if} \ \ old \leq scanned \\ Store & \textbf{otherwise} \ \ \rangle \end{cases}
\end{array}
$$

Si le mutateur vient de griser l'objet *old* (action 30) ou de constater qu'il est déjà gris (action 29), il exécute cette action, qui détermine (par une comparaison avec *scanned*) si *old* fait partie des objets déjà parcourus par le collecteur. Si c'est le cas, il faudra positionner *dirty*.

$$
\begin{aligned}
32\ \langle\ & pc = \mathit{SetDirty} \\
& \quad \overset{w}{\Longrightarrow}\ \mathit{dirty} \leftarrow \mathbf{true} \\
& \quad\quad\ \ pc \leftarrow \mathit{Store}\ \rangle
\end{aligned}
$$

Cette action positionne *dirty* si l'action 31 a déterminé que c'est nécessaire.

$$
\begin{aligned}
33\ \langle\ & pc = \mathit{Store} \\
& \quad \overset{w}{\Longrightarrow}\ \mathit{heap}[\mathit{field}] \leftarrow \mathit{new} \\
& \quad\quad\ \ pc \leftarrow \mathit{Work}\ \rangle
\end{aligned}
$$

Cette action est toujours la dernière exécutée par la procédure de modification. Elle effectue l'écriture puis sort de la procédure en retournant dans l'état normal ($pc = \mathit{Work}$).

5.2.4 Le collecteur

Contrairement aux mutateurs, presque toutes les actions du collecteur portent une contrainte d'équité faible. C'est parce que nous voulons garantir que le collecteur finira toujours par désallouer un objet devenu inaccessible, donc nous devons nous assurer qu'il ne s'arrête pas de travailler.

Variables locales

$$
\begin{aligned}
&\mathbf{type}\ \ \mathit{Steps} \triangleq \{\mathit{Sweep}, \mathit{Clear}, \mathit{Mark}, \mathit{Scan}\} \\
&\mathbf{var}\ \ \ \ \mathit{step} = \mathit{Sweep} \in \mathit{Steps}
\end{aligned}
$$

La variable *step* joue le même rôle que la variable *pc* du mutateur : elle indique quelle est l'étape courante du collecteur.

$$
\begin{aligned}
\mathbf{var}\ & \mathit{phase} = \mathit{Async} \in \mathit{Statuses} \\
& \mathit{ptr} = \mathit{limit} = \mathit{sublimit} = \mathit{rover} = 0 \in \mathit{Addr} \\
& \mathit{reset} = \mathbf{true} \in \mathit{Bool} \\
& \mathit{toWhite} = \mathit{toBlack} = \mathit{toTrace} = \emptyset \in \mathbf{set\ of}\ \mathit{Addr} \\
& \mathit{claim} = \mathit{cache} = \mathit{fields} = \emptyset \in \mathbf{multiset\ of}\ \mathit{Addr}
\end{aligned}
$$

La variable *phase* sert à gérer les poignées de mains (comme dans le pseudo-code de la section 4.4.1). Nous expliquerons le rôle des autres variables au fur et à mesure qu'elles seront utilisées par les actions.

Vue d'ensemble

Les actions du collecteur sont données dans l'ordre d'un cycle, en commençant par l'étape de balayage.

L'étape *Sweep* examine la couleur de chaque objet du tas pour prendre la décision de le blanchir ou de le désallouer (34, 35, 36). L'étape *Sweep* se termine en déclenchant l'étape *Clear* (37, 38).

Le blanchissage et la désallocation sont effectués pendant les étapes *Sweep* et *Clear* par des actions séparées (40 et 43, 44).

L'étape *Clear* repère les objets gris et noirs laissés par l'étape *Sweep* (39) et les blanchit (40). Les deux premières poignées de mains ont lieu avant le passage à l'étape *Mark* (41, 42).

La procédure *Handshake* permet au collecteur d'attendre les réponses des mutateurs (45, 46).

L'étape *Mark* réinitialise *swept* (55) et déclenche la troisième poignée de mains (56). En attendant la fin de cette poignée de mains, le collecteur marque les variables globales (47, 48, 49). Lorsque la troisième poignée de mains est finie, le collecteur passe à l'étape *Scan* (57).

Les étapes *Mark* et *Scan* utilisent la procédure *Trace* pour effectuer le parcours du graphe mémoire (50–54).

L'étape *Scan* parcourt le tas pour trouver les objets gris (61, 62). Elle doit recommencer ce parcours en cas de débordement du cache ou lorsque *dirty* est positionné par un mutateur (58, 59, 60). Cette étape se termine en déclenchant l'étape *Sweep* (63).

Balayage

Pour exécuter un pas de l'étape *Sweep*, le collecteur examine l'objet situé à l'adresse *swept*, pour le désallouer s'il est blanc et le blanchir s'il est gris ou noir. Ensuite, il avance *swept* jusqu'à l'objet suivant en ajoutant à *swept* la taille de l'objet. Il faut donc lire la taille de l'objet (dans son en-tête) et l'ajouter à *swept*. Mais le collecteur ne peut pas faire les deux dans la même action, car cela violerait la limite d'un seul accès aux variables globales par action (page 90).

Il faut donc deux actions : l'une lit l'en-tête (qui contient la taille et la couleur) de l'objet, et l'autre incrémente *swept*. Il faut aussi une variable locale au mutateur pour transmettre la taille de l'objet entre ces deux actions. À la place de la taille de l'objet, nous utilisons une information équivalente : l'adresse de l'objet suivant (*ptr*). Ainsi, c'est la première action qui effectue l'addition (taille + *swept*), mais c'est la deuxième qui stocke le résultat dans *swept*.

Nous avons donc la variable *ptr* qui contient toujours l'adresse du prochain objet que le collecteur va examiner et la variable globale *swept* qui est toujours "approximativement égale" à *ptr*. Les mutateurs consultent *swept* pour décider de la couleur à donner aux objets nouvellement alloués (action 22).

Par "approximativement égale", nous entendons que l'un des trois cas suivants est vrai :

- $swept = ptr$
- la variable *swept* pointe sur l'en-tête ou sur un champ d'un objet, et *ptr* pointe sur l'en-tête de l'objet suivant
- les objets inclus dans l'intervalle $[swept, ptr[$ et l'objet sur lequel pointe *swept* formaient un objet libre qui vient d'être découpé (une ou plusieurs fois) par un ou plusieurs mutateurs.

$$
\begin{aligned}
34\ \langle\ & step = Sweep\ \wedge\ swept = ptr < sublimit \\
& \stackrel{w}{\Longrightarrow} \textbf{if}\ \ heap[ptr].\,color \in \{Gray, Black\}\ \textbf{then}\ \ toWhite \leftarrow toWhite \cup \{ptr\} \\
& \quad\ \textbf{else if}\ \ heap[ptr].\,color = White\ \ \textbf{then}\ \ claim \leftarrow claim \oplus \{ptr\} \\
& \quad\ ptr \leftarrow ptr + heap[ptr].\,size + 1\ \rangle
\end{aligned}
$$

Cette action examine la couleur de l'objet courant et détermine ce que le collecteur fera de cet objet. Si l'objet est noir ou gris, il est placé dans l'ensemble *toWhite* des objets à blanchir ; s'il est blanc il est placé dans l'ensemble *claim* des objets à désallouer, et sinon il est bleu et le collecteur se contente de l'ignorer. L'objet étant traité, on avance *ptr* jusqu'à l'objet suivant du tas. La précondition $swept = ptr$ assure que *ptr* ne prend pas plus d'un objet d'avance sur *swept*.

Le rôle de *sublimit* sera décrit avec l'action 36.

$$
\begin{aligned}
35\ \langle\ & step = Sweep\ \wedge\ swept < ptr \\
& \stackrel{w}{\Longrightarrow} swept \leftarrow ptr\ \rangle
\end{aligned}
$$

Cette action permet à *swept* de rattraper *ptr* et rétablit donc la précondition de l'action 34.

$$
\begin{aligned}
36\ \langle\ & step = Sweep\ \wedge\ ptr < x \leq limit \\
& \stackrel{w}{\Longrightarrow} free \leftarrow free \ominus \{ptr, \ldots, x-1\} \\
& \quad\ sublimit \leftarrow x\ \rangle
\end{aligned}
$$

Quand le collecteur trouve un objet blanc, il le recolle avec l'objet libre précédent (action 43) ou il le place dans *free* (action 44). Mais il y a déjà des objets dans *free* avant le début du balayage, et ils sont blancs. Le collecteur va donc les trouver et les rendre invalides en les recollant (ou les placer dans *free*). Pour éviter qu'un objet de *free* soit invalide ou se trouve en plusieurs exemplaires dans *free* (et soit donc alloué plusieurs fois), nous empêchons l'action 34 de voir des objets blancs de *free*. Pour cela, nous limitons l'action 34 pour qu'elle ne dépasse pas *sublimit* et nous retirons de *free* tous les objets blancs compris entre *ptr* et *sublimit*. L'action 36 avance *sublimit* en choisissant une adresse x comprise entre *ptr* et *limit*, qui sera la nouvelle valeur de *sublimit*, et en retirant de *free* tous les objets compris entre *ptr* et *sublimit*.

Nous utilisons cette technique au lieu de retirer les objets de *free* un à un ou de tester chaque objet pour savoir s'il est dans *free* avant de le placer dans *claim* car l'accès à *free* se fait dans une section critique, donc nous voulons amortir le coût de la primitive de synchronisation.

$$
\begin{aligned}
37\ \langle\ & step = Sweep\ \wedge\ ptr = limit \\
& \stackrel{w}{\Longrightarrow} swept \leftarrow +\infty\ \rangle \\
38\ \langle\ & step = Sweep\ \wedge\ swept = +\infty \\
& \stackrel{w}{\Longrightarrow} limit \leftarrow top \\
& \quad\ step \leftarrow Clear\ \rangle
\end{aligned}
$$

La variable *limit* marque la borne supérieure de la zone du tas qui sera examinée par le balayage. Le balayage ne se fait pas jusqu'à la limite supérieure *top* du tas, car celle-ci peut changer pendant le balayage. Si *top* augmente plus vite que *ptr*, le balayage ne terminera jamais et le GC ne pourra donc plus désallouer les objets morts. Pour éviter ce problème, nous copions la valeur de *top* dans *limit* avant le début du balayage, et celui-ci s'arrête lorsqu'il rejoint *limit*.

L'action 37 donne la valeur $+\infty$ à *swept*, ce qui fait que les mutateurs ne créeront plus que des objets blancs (action 22). La valeur de *top* à cet instant marque donc la limite à partir de laquelle il n'y a plus d'objets noirs (à part les objets en cours de création).

L'étape *Clear* va blanchir les objets noirs restants pour préparer le cycle suivant. L'action 38 copie la valeur courante de *top* dans *limit* et passe à l'étape *Clear*. À partir de cet instant, tous les objets noirs restants se trouvent soit dans l'ensemble *toWhite* (et le GC va les blanchir) soit entre *ptr* et *limit*. En effet, les objets situés avant *ptr* ont été balayés par l'étape *Sweep*, et les objets situés après *limit* ont été alloués alors que *swept* valait $+\infty$. On évite ainsi à l'étape *Clear* d'avoir à balayer jusqu'à *top*.

L'étape *Clear*

39 $\langle\ step = Clear\ \wedge\ ptr < limit$
$\quad\stackrel{w}{\Longrightarrow}$ **if** $heap[ptr].\,color \in \{Gray, Black\}$ **then** $toWhite \leftarrow toWhite \cup \{ptr\}$
$\qquad ptr \leftarrow ptr + heap[ptr].\,size + 1\ \rangle$

Cette action est une version simplifiée de l'action 34 : on n'utilise plus *sublimit*, on n'a pas besoin de mettre à jour *swept* et on ne désalloue pas les objets blancs.

40 $\langle\ x \in toWhite$
$\quad\stackrel{w}{\Longrightarrow} toWhite \leftarrow toWhite \setminus \{x\}$
$\qquad heap[x].\,color \leftarrow White\ \rangle$

Cette action est en fait commune aux étapes *Sweep* et *Clear* : elle blanchit les objets que le collecteur a décidé de blanchir.

41 $\langle\ step = Clear\ \wedge\ ptr = limit\ \wedge\ toWhite = \emptyset$
$\quad\stackrel{w}{\Longrightarrow} status_C \leftarrow Sync_1\ \rangle$

Cette action démarre la première poignée de mains en changeant $status_C$. La première poignée de mains ne peut démarrer que lorsque le GC a fini de blanchir tous les objets noirs. En effet, le début de la première poignée de mains signale aux mutateurs qu'il faut griser les objets lors des modifications. Il ne faut pas que ces objets gris soient ensuite blanchis par le GC. La fin de cette poignée de mains sera gérée par l'action 45.

42 $\langle\ step = Clear\ \wedge\ phase = Sync_1\ \wedge\ claim = \emptyset$
$\quad\stackrel{w}{\Longrightarrow} status_C \leftarrow Sync_2$
$\qquad step \leftarrow Mark\ \rangle$

Cette action marque le début de la deuxième poignée de mains et le démarrage de l'étape *Mark*. La condition $phase = Sync_1$ indique que la première poignée de mains est finie. La condition $claim = \emptyset$ oblige le collecteur à terminer ses désallocations avant de démarrer un nouveau cycle.

Désallocation

Nous avons deux actions de désallocation. Ces actions s'exécutent en parallèle avec les étapes *Sweep* et *Clear*, mais pas avec les autres (car $claim = \emptyset$ pendant les étapes *Mark* et *Scan*).

$$43\ \langle\ x \in claim\ \wedge y = x + heap[x].\,size + 1 \in claim \\ \Longrightarrow claim \leftarrow claim \ominus \{y\} \\ heap[x] \leftarrow \ \mathbf{record} \begin{cases} color \mapsto White \\ size \ \ \mapsto heap[x].\,size + heap[y].\,size + 1\ \rangle \end{cases}$$

Cette action permet au GC de recoller deux objets libres adjacents pour en faire un seul objet plus gros. Elle est indispensable pour lutter contre la fragmentation. Les objets libres ainsi recollés ne doivent pas être dans la liste libre car nous ne voulons pas imposer le surcoût d'une synchronisation à cette action.

$$44\ \langle\ x \in claim \\ \stackrel{w}{\Longrightarrow} claim \leftarrow claim \ominus \{x\} \\ free \leftarrow free \oplus \{x\}\ \rangle$$

Cette action est la raison d'être du GC : c'est l'action qui désalloue (en les plaçant dans la liste libre) les objets que le GC a déterminé inaccessibles.

Poignées de mains

$$45\ \langle\ status_C \neq phase\ \wedge \forall m \in Pid, status_m \neq phase \\ \stackrel{w}{\Longrightarrow} phase \leftarrow status_C\ \rangle$$

Cette action correspond à la deuxième phase de la procédure *Handshake* : elle attend qu'une poignée de mains soit commencée ($status_C \neq phase$) et que tous les mutateurs aient répondu ($\forall m \in Pid, status_m \neq phase$), et elle termine alors la poignée de mains en avançant *phase*. Les autres actions du collecteur commencent une poignée de mains en changeant $status_C$ et détectent la fin de la poignée de mains lorsque $phase = status_C$.

Dans l'implémentation, le GC maintient deux listes de processus : ceux qui n'ont pas encore répondu et ceux qui ont déjà répondu. Le GC teste les processus l'un après l'autre et les fait passer de la première à la deuxième liste lorsqu'ils répondent. Le test $\forall m \in Pid, status_m \neq phase$ est alors vrai lorsque la première liste est vide.

$$46\ \langle\ status_m = Dead \\ \stackrel{w}{\Longrightarrow} status_m \leftarrow Avail\ \rangle$$

Par cette action, le GC libère les processus morts pour qu'ils puissent être recyclés par les actions de lancement de processus. Il peut alors les retirer de la liste des processus.

Variables globales

Les variables globales ne sont pas représentées en tant que telles dans notre modèle. On peut les coder par un processus dont les racines sont les variables globales et qui ne fait rien d'autre que répondre aux poignées de mains et marquer ses variables globales. Ainsi, les instructions du pseudo-code qui marquent les variables globales sont modélisées ici par ce processus supplémentaire.

Il faut encore que le collecteur puisse trouver ces objets pour les noircir et marquer leurs fils. Plus généralement, notre modèle permet au GC de trouver par un moyen quelconque un objet gris dans le tas pour le noircir. Ce moyen de trouver les objets gris pourrait être une structure de données dans laquelle les mutateurs écrivent des adresses d'objets gris, sans utiliser de synchronisation. Un tel canal de communication n'est pas sûr : il peut perdre des adresses (si deux mutateurs écrivent au même endroit, par exemple), mais il donne au collecteur un moyen peu coûteux de trouver la plupart des objets gris, donc de limiter le nombre de parcours effectués par l'étape *Scan*.

47 $\langle$ $step \in \{Mark, Scan\}$
$\quad\Longrightarrow rover \leftarrow 0 \rangle$
48 $\langle$ $step \in \{Mark, Scan\} \ \wedge \ rover < top$
$\quad\Longrightarrow rover \leftarrow rover + heap[rover].\,size + 1 \rangle$
49 $\langle$ $step \in \{Mark, Scan\} \ \wedge \ rover < top \ \wedge \ heap[rover].\,color = Gray$
$\quad\Longrightarrow toBlack \leftarrow toBlack \cup \{rover\} \rangle$

Une adresse de la mémoire qui contient un en-tête ne représente pas forcément un objet valide. Les objets valides sont ceux qui font partie du pavage du tas. On les trouve en partant de l'objet situé à l'adresse 0 et en ajoutant à chaque fois la taille de l'objet à son adresse, comme le fait le code du balayage. C'est ce que nous faisons ici avec la variable *rover* : nous pouvons à tout moment commencer un nouveau parcours, avancer le parcours courant, ou constater que l'objet courant est gris et le placer dans l'ensemble des objets à noircir.

Ces actions ne comportent pas de contrainte d'équité car elles sont optionnelles : une implémentation n'est pas obligée d'utiliser une variable *rover*. Le collecteur peut à tout moment trouver un en-tête d'objet gris valide dans le tas et placer cet objet dans *toBlack*. La façon de trouver cet objet n'a pas d'importance, du moment que c'est un objet valide. Ces trois actions servent en fait à définir précisément la notion d'objet "valide" sans faire intervenir une définition auxiliaire compliquée. On ne retrouvera sans doute pas ces actions dans une implémentation de notre algorithme, car le parcours avec *rover* à partir du début du tas ferait double emploi avec le parcours de l'étape *Scan*.

La procédure *Trace*

La procédure *Trace* effectue l'essentiel du travail des étapes *Mark* et *Scan* : c'est elle qui parcourt le graphe mémoire pour marquer tous les objets accessibles. On l'active en plaçant un objet dans l'ensemble *toBlack*.

$$
\begin{array}{l}
50 \ \langle\ x \in \mathit{toBlack} \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{heap}[x].\,\mathit{color} \leftarrow \mathit{Black} \\
\qquad\qquad \mathit{toBlack} \leftarrow \mathit{toBlack} \setminus \{x\} \\
\qquad\qquad \mathit{cache} \leftarrow \mathit{cache} \oplus \{x\}\ \rangle
\end{array}
$$

Cette action retire un objet de *toBlack*, le noircit et le place dans le cache. Dans une implémentation normale (comme le pseudo-code du chapitre 4), l'ensemble *toBlack* contient au maximum un élément : l'objet en cours de traitement par la procédure *Trace*.

$$
\begin{array}{l}
51 \ \langle\ x \in \mathit{cache}\ \wedge\ \mathit{cache} \neq \{x\} \\
\qquad \Longrightarrow \mathit{cache} \leftarrow \mathit{cache} \ominus \{x\} \\
\qquad\qquad \mathit{heap}[x].\,\mathit{color} \leftarrow \mathit{Gray} \\
\qquad\qquad \mathbf{if}\ x < \mathit{ptr}\ \ \mathbf{then}\ \ \mathit{reset} \leftarrow \mathbf{true}\ \rangle
\end{array}
$$

Cette action gère le débordement du cache : le collecteur peut à tout moment retirer un objet du cache et le recolorier en gris. La condition $\mathit{cache} \neq \{x\}$ assure que le cache ne deviendra pas vide s'il y a du travail à faire. Avec la condition d'équité faible sur l'action 52, cette condition assure que le parcours du graphe mémoire ne peut pas rester bloqué.

Si l'objet que l'on sort du cache se trouve dans la partie du tas qui a déjà été parcourue, il faut forcer le collecteur à effectuer un nouveau parcours. La variable *reset* est un reflet local de la variable globale *dirty*. Son rôle est expliqué en détail avec l'action 58.

$$
\begin{array}{l}
52 \ \langle\ x \in \mathit{cache} \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{cache} \leftarrow \mathit{cache} \ominus \{x\} \\
\qquad\qquad \mathit{fields} \leftarrow \mathit{fields} \oplus \{x+1, \ldots, x + \mathit{heap}[x].\,\mathit{size}\}\ \rangle
\end{array}
$$

Cette action retire un objet du cache et initialise la boucle qui énumère ses champs. Le travail qui reste à faire par cette boucle est représenté par l'ensemble *fields* qui contient les adresses des champs restant à examiner. Nous ne pouvons pas placer directement les valeurs des champs car il faudrait pour cela effectuer plusieurs lectures de *heap* dans une action atomique.

$$
\begin{array}{l}
53 \ \langle\ x \in \mathit{fields} \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{fields} \leftarrow \mathit{fields} \ominus \{x\} \\
\qquad\qquad \mathit{toTrace} \leftarrow \mathit{toTrace} \cup \{\mathit{heap}[x]\}\ \rangle
\end{array}
$$

Cette action est le corps de la boucle qui énumère les champs de l'objet en cours de traitement. Elle retire une adresse x de l'ensemble des champs qui restent à examiner et elle lit la valeur de ce champ. Elle place cette valeur dans l'ensemble *toTrace*, qui est l'ensemble des objets dont il faudra tester la couleur et qu'il faudra noircir si nécessaire.

$$
\begin{array}{l}
54 \ \langle\ x \in \mathit{toTrace} \\
\qquad \stackrel{w}{\Longrightarrow} \mathit{toTrace} \leftarrow \mathit{toTrace} \setminus \{x\} \\
\qquad\qquad \mathbf{if}\ \ \mathit{heap}[x].\,\mathit{color} \in \{\mathit{White}, \mathit{Gray}\}\ \mathbf{then}\ \ \mathit{toBlack} \leftarrow \mathit{toBlack} \cup \{x\}\ \rangle
\end{array}
$$

Cette action examine la couleur d'un objet de *toTrace*, et le place dans *toBlack* s'il y a lieu. Placer l'objet dans *toBlack* correspond à l'appel récursif de la procédure *Trace* dans le pseudo-code.

L'étape *Mark*

55 $\langle$ $phase \neq Async$
$\quad\quad \overset{w}{\Longrightarrow} swept \leftarrow -\infty \rangle$
56 $\langle$ $step = Mark \ \wedge\ phase = Sync_2 \ \wedge\ swept = -\infty$
$\quad\quad \overset{w}{\Longrightarrow} status_C \leftarrow Async \rangle$

La variable *swept* doit rester égale à $+\infty$ jusqu'à la fin de la première poignée de mains. Lorsque la première poignée de mains est finie, les mutateurs savent tous que l'étape *Clear* est finie, donc ils ne consultent plus la valeur de *swept*. Ils recommenceront à la consulter après avoir répondu à la troisième poignée de mains (pour savoir si l'étape *Sweep* est commencée, dans les actions 22 et 28). Il faut donc changer *swept* de $+\infty$ à $-\infty$ entre la fin de la première poignée de mains et le début de la troisième. L'action 55 change *swept* après la fin de la première poignée de mains, et l'action 56 ne déclenche la troisième poignée de mains que lorsque *swept* a changé.

57 $\langle$ $step = Mark \ \wedge\ phase = Async$
$\quad\quad \overset{w}{\Longrightarrow} ptr \leftarrow limit \leftarrow top$
$\quad\quad\quad step \leftarrow Scan \rangle$

Lorsque la troisième poignée de mains est finie, cette action initialise l'étape *Scan*. Tous les objets situés au-delà de *limit* pendant l'étape *Scan* seront créés en noir (car $swept = -\infty$). Il est donc inutile de les parcourir pour chercher les objets gris. L'étape *Scan* se contentera donc de parcourir l'intervalle $[0, limit[$.

Démarrage du parcours (étape *Scan*)

Lorsque le collecteur décide de démarrer ou de redémarrer un parcours du tas, il positionne la variable *reset*. Cela arrive quand il y a débordement du cache (action 51) ou quand le collecteur constate qu'un mutateur a positionné *dirty*. Nous utilisons la variable locale *reset* pour matérialiser la décision de refaire un parcours car le collecteur ne peut pas dans la même action tester *dirty* et changer *scanned*, car ce sont deux variables globales.

58 $\langle$ $reset \ \vee\ ptr = limit \ \wedge\ cache = fields = toBlack = toTrace = \emptyset$
$\quad\quad \overset{w}{\Longrightarrow} \textbf{if } step = Scan \textbf{ then } ptr \leftarrow limit$
$\quad\quad\quad scanned \leftarrow -\infty \rangle$
59 $\langle$ $step = Scan \ \wedge\ reset \ \wedge\ scanned = -\infty$
$\quad\quad \overset{w}{\Longrightarrow} ptr \leftarrow 0$
$\quad\quad\quad reset \leftarrow dirty \leftarrow \textbf{false} \rangle$

Le (re)démarrage du parcours se fait en deux actions car il faut changer deux variables globales : *scanned* et *dirty*. L'action 58 initialise *scanned*. Elle est activée lorsque le collecteur décide de démarrer un nouveau parcours (*reset* est vrai) ou si le dernier parcours est fini ($ptr = limit$ et tous les ensembles sont vides), pour préparer le premier parcours du cycle suivant. L'action 59 initialise *dirty* et *reset* et remet *ptr* à zéro.

60 $\langle\ step = Scan\ \wedge\ dirty\ \wedge (scanned < ptr\ \vee\ ptr = limit)$
$\quad\quad\stackrel{w}{\Longrightarrow} reset \leftarrow \mathbf{true}\ \rangle$

Par cette action, le collecteur constate que *dirty* est vrai et décide de faire un nouveau parcours. Le collecteur peut prendre cette décision même si le parcours courant n'est pas fini, donc notre modèle a plus de souplesse que le pseudo-code, qui ne prend cette décision qu'à la fin d'un parcours. Cette souplesse peut se traduire par une plus grande efficacité : si le parcours vient de commencer et si *dirty* est déjà positionné, on peut économiser la fin du parcours car il suffit de le stopper et d'en recommencer un autre.

La condition $scanned < ptr\ \vee\ ptr = limit$ oblige le collecteur à parcourir l'objet qui a déclenché le changement de *dirty*. En effet, les actions 31 et 32 du mutateur positionnent *dirty* si l'objet *scanned* est gris (car le mutateur ne sait pas si cet objet a déjà été parcouru ou non). Si le collecteur ne parcourt pas cet objet avant de recommencer son parcours, l'objet reste gris, et le mutateur peut à nouveau positionner *dirty* lors du parcours suivant au moment où le collecteur va examiner le même objet. Le mutateur pourrait ainsi forcer le collecteur à recommencer son parcours indéfiniment.

Parcours du tas (étape *Scan*)

Le parcours du tas se fait avec les variables *scanned* et *ptr*, qui sont utilisées de la même façon que *swept* et *ptr* pour le balayage (actions 34 et 35).

61 $\langle\ step = Scan\ \wedge\ scanned < ptr\ \wedge \neg(reset\ \wedge\ scanned = -\infty)$
$\quad\quad\stackrel{w}{\Longrightarrow} scanned \leftarrow ptr\ \rangle$

Nous commençons par l'action qui permet à *scanned* de rattraper *ptr*, puisque le parcours démarre avec $scanned = -\infty$ et $ptr = 0$. De même que pour le balayage, il doit y avoir au maximum un objet entre *scanned* et *ptr*, exception faite des objets en cours de création. Ceux-ci posent moins de problèmes que pour le balayage, puisque les objets créés sont tous noirs pendant l'étape *Scan*.

Cette action ne peut pas se déclencher quand $reset\ \wedge\ scanned = -\infty$ car cette condition correspond à l'état transitoire du redémarrage du parcours, entre l'exécution de l'action 58 et celle de l'action 59.

62 $\langle\ step = Scan\ \wedge\ scanned = ptr < limit$
$\quad\quad\stackrel{w}{\Longrightarrow} \mathbf{if}\ \ heap[ptr].\,color = Gray\ \ \mathbf{then}\ \ toBlack \leftarrow toBlack \cup \{ptr\}$
$\quad\quad\quad\quad ptr \leftarrow ptr + heap[ptr].\,size + 1\ \rangle$

Cette action avance *ptr* en testant la couleur de l'objet courant et en plaçant celui-ci dans l'ensemble *toBlack* s'il est gris. Son fonctionnement est analogue à celui de l'action 34.

$$
\begin{array}{l}
63\ \langle\ step = Scan\ \wedge\ ptr = limit\ \wedge \neg\, reset\ \wedge \neg\, dirty \\
\qquad \wedge\ cache = fields = toBlack = toTrace = \emptyset \\
\qquad\quad \stackrel{w}{\Longrightarrow} reset \leftarrow \mathbf{true} \\
\qquad\qquad\quad rover \leftarrow ptr \leftarrow sublimit \leftarrow 0 \\
\qquad\qquad\quad step \leftarrow Sweep\ \rangle
\end{array}
$$

Cette action marque la fin de l'étape *Scan* et le début de l'étape *Sweep*. L'affectation $reset \leftarrow$ **true** initialise l'étape *Scan* du cycle suivant. Nous remettons *rover* à zéro car l'étape *Sweep* risque de recoller des objets (action 43) et de rendre *rover* invalide.

Nous remettons *ptr* et *sublimit* à zéro pour initialiser l'étape *Sweep* et nous lançons cette étape en changeant *step*.

Chapitre 6

Preuve du GC concurrent

Ce chapitre donne une description de la preuve de correction de l'algorithme exposé dans le chapitre 5. La preuve de correction repose sur la conjonction de 46 invariants. On prouve que tous ces invariants sont conservés par chacune des 63 actions : en supposant que tous les invariants sont vrais avant l'action, on prouve que chacun est encore vrai après. On prouve aussi que tous les invariants sont satisfaits par les conditions initiales. On en déduit que les invariants sont vrais à tout instant dans toutes les exécutions possibles du programme.

La section 6.1 explique les définitions des fonctions, ensembles et relations auxiliaires utilisés dans l'écriture des invariants. La section 6.2 expose les invariants. La section 6.3 donne les grandes lignes de la preuve.

6.1 Définitions préliminaires

Nous définissons quelques *fonctions d'état*. Ce sont des expressions dont la valeur dépend des variables du programme, donc de l'état considéré. Elles nous serviront à définir les invariants (qui sont des prédicats sur les états).

$$\mathsf{Af}(x) \qquad \triangleq x + \mathit{heap}[x].\,\mathit{size} + 1$$

Af est une fonction qui, étant donné l'adresse x d'un objet du tas, donne l'adresse du premier mot situé après cet objet. L'en-tête de l'objet est à l'adresse x et ses champs sont situés aux adresses $x \ldots x + \mathit{heap}[x].\,\mathit{size}$. Le premier mot après cet objet est donc à l'adresse $x + \mathit{heap}[x].\,\mathit{size} + 1$. En général, on trouve à cet endroit l'objet suivant (dans l'ordre des adresses croissantes).

Si x n'est pas l'adresse d'un en-tête, la valeur de Af est quelconque, nous ne la définissons pas (et nous ne l'utiliserons bien sûr pas dans la preuve).

$$x \mathsf{A} y \qquad \triangleq x < \mathit{top} \ \wedge\ \mathit{heap}[x] \in \mathit{Headers} \ \wedge y = \mathsf{Af}(x)$$

A est la relation d'adjacence des objets du tas. Elle relie x et y si x est l'adresse d'un en-tête situé dans le tas et si y est l'adresse qui suit l'objet situé à l'adresse x.

$$\mathsf{Obj} \quad \triangleq \mathsf{A}^*(0) \cap [0, top[$$

Obj est l'ensemble de tous les objets chaînés du tas. Un objet *chaîné* est un objet que l'on peut atteindre en itérant la relation A à partir de 0. Nous nous restreignons à l'intervalle $[0, top[$, qui représente la partie utile de la mémoire (le tas). Aucun objet utilisé par les mutateurs ne se trouve au-delà de top. Dans la suite, nous écrirons simplement "objet" pour parler des objets chaînés. Nous verrons de plus que le "suivant" du dernier objet est toujours exactement top.

$$\begin{array}{ll}
\mathsf{Wh} & \triangleq \{x \in \mathsf{Obj} \mid heap[x].\,color = White\} \\
\mathsf{Gr} & \triangleq \{x \in \mathsf{Obj} \mid heap[x].\,color = Gray\} \\
\mathsf{Bk} & \triangleq \{x \in \mathsf{Obj} \mid heap[x].\,color = Black\} \\
\mathsf{Bu} & \triangleq \{x \in \mathsf{Obj} \mid heap[x].\,color = Blue\}
\end{array}$$

Wh, Gr, Bk et Bu sont respectivement l'ensemble des objets blancs, gris, noirs et bleus.

$$\mathsf{FreeList}_m \quad \triangleq pool_m \oplus \begin{cases} \{old_m\} & \textbf{if } pc_m = Split \\ \emptyset & \textbf{otherwise} \end{cases}$$

$\mathsf{FreeList}_m$ est la liste libre locale du mutateur m, à laquelle on ajoute old_m lorsque le mutateur est sur le point d'exécuter l'action 21. En effet, si $pc_m = Split$, le mutateur a provisoirement retiré old_m de sa liste libre locale (à cause de l'action 19 qui peut prendre un objet de la liste libre locale pour le remettre dans la liste libre globale). $\mathsf{FreeList}_m$ représente donc l'ensemble des objets bleus qui sont réservés par le mutateur m.

$$\mathsf{Unswept} \quad \triangleq \begin{cases} [ptr, limit[& \textbf{if } step = Sweep \\ \emptyset & \textbf{otherwise} \end{cases}$$

Lorsque le GC est dans l'étape *Sweep*, Unswept représente la partie du tas qui va être balayée. Cette partie contient tous les objets sur le point d'être désalloués.

$$\mathsf{Fr} \quad \triangleq (\mathsf{Wh} \cap \mathsf{Unswept}) \cup claim \cup free \cup \mathsf{Bu}$$

Fr est l'ensemble des objets :

- que le GC va balayer, trouver blancs et désallouer,
- que le GC a déjà décidé de désallouer,
- que le GC a déjà désalloué (première partie de la liste libre globale) ou
- qui sont bleus (le reste de la liste libre globale et les listes libres locales).

Les éléments de Fr sont donc les objets qui sont déjà dans une liste libre ou qui vont bientôt y être.

$$\mathsf{Creating}_m \quad \triangleq \begin{cases} \{new_m\} & \textbf{if } pc_m \in CreateProc \setminus \{Split\} \\ \emptyset & \textbf{otherwise} \end{cases}$$

$\mathsf{Creating}_m$ est l'ensemble des objets qui sont en cours de création ou de remplissage par le mutateur m. Cet ensemble ne peut pas avoir plus d'un élément. $\mathsf{Creating}_m$ est vide si $pc_m = Split$ car dans ce cas new_m n'est pas l'adresse d'un objet chaîné : c'est encore un pointeur infixe dans l'objet old_m, qui n'a pas encore été découpé.

$$\mathsf{Val} \quad \triangleq \mathsf{Obj} \setminus (\mathsf{Fr} \cup \bigcup \mathsf{Creating}_{Pid})$$

Val est l'ensemble des objets *valides* : c'est l'ensemble des objets qui ne sont pas libres et qui ne sont pas en cours de création ni de remplissage. Les éléments de Val sont donc les objets dont le contenu est significatif : ils ont été alloués et initialisés, et ils ne sont pas considérés comme morts par le GC. Celui-ci ne changera pas leur contenu. Val est donc l'ensemble des nœuds du graphe mémoire.

$$\mathsf{Cf}(x) \quad \triangleq \,]x, \mathsf{Af}(x)[$$

Si x est l'adresse d'un objet, $\mathsf{Cf}(x)$ est l'ensemble des adresses des champs de cet objet.

$$x\mathsf{C}y \quad \triangleq x \in \mathsf{Val} \ \wedge \ y \in \mathsf{Cf}(x)$$

La relation C relie chaque objet valide à tous ses champs.

$$x\mathsf{Hp}y \quad \triangleq x \in \mathsf{C}(\mathsf{Val}) \ \wedge \ y = heap[x]$$

Hp relie x et y si x est l'adresse d'un champ d'un objet valide qui pointe sur y.

$$x\mathsf{P}y \quad \triangleq x \in \mathsf{Val} \ \wedge \ y \in \mathsf{Hp}(\mathsf{C}(x))$$

P relie x et y si x est l'adresse d'un objet valide qui contient parmi ses champs un pointeur égal à y. On a donc $x\mathsf{P}y$ si x pointe vers y (si y est un fils de x). P est la relation qui représente les flèches du graphe mémoire.

$$\mathsf{AccField}_m \quad \triangleq \{heap[x] \mid pc_m \in CreateProc \wedge x \in \mathsf{Cf}(new_m) \setminus toFill_m\} \setminus \{new_m\}$$

Lorsque le mutateur m est en train de créer un objet (et de le remplir), $\mathsf{AccField}_m$ est l'ensemble des pointeurs que le mutateur a déjà placés dans les champs de l'objet new_m. Ce sont des pointeurs qui ne font pas encore partie du graphe mémoire, mais qui en feront partie dès que le mutateur aura fini le remplissage de l'objet new_m et placé cet objet dans ses racines. Il est donc important de s'assurer que ces pointeurs resteront valides. Le mutateur peut placer des pointeurs vers l'objet new_m dans les champs de new_m. Nous excluons ces pointeurs de $\mathsf{AccField}_m$ car ils ne sont pas encore valides.

$$\mathsf{AccUpd}_m \quad \triangleq \begin{cases} \{new_m, old_m\} \cup \{x \in \mathsf{Val} \mid x\mathsf{C}\,field_m\} & \textbf{if } pc_m \in UpdateProc \\ \emptyset & \textbf{otherwise} \end{cases}$$

Lorsque le mutateur m est en train d'exécuter sa procédure de modification du graphe mémoire, AccUpd_m est l'ensemble des objets dont le mutateur m est susceptible de changer la couleur (new_m et old_m) ou le contenu (l'objet qui contient $field_m$). Il est important que ces objets restent valides au moins jusqu'à la fin de la procédure de modification.

$$\mathsf{AccArg}_m \quad \triangleq \begin{cases} \widehat{args}_{child_m} & \textbf{if } pc_m = Launch \\ \emptyset & \textbf{if } pc_m \neq Launch \wedge status_m = Quick \\ \widehat{args}_m & \textbf{otherwise} \end{cases}$$

AccArg_m est l'ensemble des arguments dont le mutateur m est “responsable”. Nous attribuons au processus qui est en train d'en lancer un autre (le *père*) la responsabilité des arguments du nouveau processus (le *fils*), tant que le père est en train d'accumuler les arguments du fils. Le fils n'est donc responsable d'aucun argument tant que son *status* est *Quick*. Tous les autres processus sont responsables de leurs propres arguments. En général, ils n'en ont pas car $args_m$ est vide lorsque $pc_m \neq Halt$.

Comme $\mathsf{AccField}_m$, l'ensemble AccArg_m contient des pointeurs qui ne font pas encore partie du graphe mémoire, mais qui ne vont pas tarder à devenir des racines.

$$\mathsf{Acc}_m \quad \triangleq \widehat{roots}_m \cup \widehat{toMark}_m \cup \mathsf{AccArg}_m \cup \mathsf{AccField}_m \cup \mathsf{AccUpd}_m$$

Acc_m représente l'ensemble des *racines étendues* de m : c'est l'ensemble des adresses de tous les objets dont le mutateur m est susceptible de lire ou d'écrire la couleur ou un champ.

Acc_m est une fonction d'état, donc, pour un état donné, c'est l'ensemble des objets que le mutateur m pourra lire ou écrire *par la prochaine action*. Cela ne préjuge en rien des actions suivantes, puisque l'état courant (et donc Acc_m) sera changé par cette prochaine action.

$$\mathsf{Kill}_m \quad \triangleq \begin{cases} \{field_m\} & \textbf{if} \quad \wedge\ pc_m \in UpdateProc \wedge status_m \neq Sync_1 \\ & \qquad \wedge\ (old_m \in \mathsf{Hp}(field_m) \Rightarrow pc_m = Store) \\ \emptyset & \textbf{otherwise} \end{cases}$$

Kill_m est un ensemble (singleton ou ensemble vide) qui contient l'adresse de la mémoire que le mutateur m est susceptible de modifier, empêchant ainsi le collecteur de voir le pointeur stocké à cette adresse. Lors d'une modification du graphe mémoire, le pointeur qui est sur le point d'être effacé (l'ancienne valeur) sera donc dans cet ensemble. Nous dirons que cet élément est *volatil*.

Si $status_m = Sync_1$, alors la deuxième poignée de mains n'est pas encore commencée. Le collecteur n'est donc pas encore en train de parcourir le graphe mémoire, donc l'affectation ne peut pas interférer avec ce parcours. Si $old_m \notin \mathsf{Hp}(fields_m)$ alors un autre mutateur a changé cette valeur après que l'action 27 l'a lue, et les actions 29 à 32, qui visent à assurer que l'ancienne valeur sera bien parcourue par le collecteur, se feront en pure perte (puisqu'elles agissent sur old_m, qui n'est plus l'ancienne valeur).

Dans ce cas, il faut considérer que $field_m$ est volatil pendant toute la procédure *UpdateProc*, puisque le mutateur va l'effacer sans agir sur l'ancienne valeur, comme s'il effectuait l'affectation sans précautions particulières. Dans le cas contraire, field_m n'est volatil que lorsque le mutateur est sur le point d'effectuer son écriture (donc lorsque $pc_m = \mathit{Store}$).

$$\mathsf{Done} \qquad \triangleq \mathsf{Bk} \backslash \widehat{cache}$$

Done est l'ensemble des objets "parcourus" (selon la terminologie de la section 3.1). Ce sont les objets noirs qui ne sont pas dans le cache (les objets du cache sont encore dans l'ensemble des objets "marqués"). Les éléments de Done sont donc les objets que l'étape de marquage a fini d'examiner.

$$\mathsf{FDone} \qquad \triangleq \mathsf{C}(\mathsf{Done} \cap \mathsf{Val}) \backslash \widehat{\mathit{fields}}$$

FDone est l'ensemble des champs parcourus : ce sont les champs des objets de Done, sauf ceux qui sont encore dans *fields* (le GC est sur le point de les parcourir).

$$x \,\mathsf{Tr}\, y \qquad \triangleq x \in \mathsf{Val} \backslash \mathsf{Done} \;\wedge\; y \in \mathsf{Hp}(\mathsf{Cf}(x) \backslash \bigcup \mathsf{Kill}_{Pid})$$

La relation Tr est la relation de traçage : l'ensemble des arêtes du graphe mémoire que le GC suivra si les mutateurs n'interfèrent pas avec son travail. C'est l'ensemble des pointeurs contenus dans les champs non volatils des objets valides non parcourus. Les pointeurs contenus dans les champs volatils ne font pas partie de cette relation car il n'y a aucun moyen de savoir s'ils vont être effacés avant que le GC les trouve. Nous les considérons comme déjà effacés (pour obtenir une relation que l'action 33 ne diminue pas, et qui est donc un sous-ensemble de la "vraie" relation de traçage).

$$\mathsf{ReachMark}(m) \triangleq \widehat{toMark}_m \cup \begin{cases} \mathsf{AccField}_m & \textbf{if } \mathit{marking}_m \\ \emptyset & \textbf{otherwise} \end{cases}$$

ReachMark(m) est l'ensemble des objets que le mutateur m va marquer (par sa procédure de marquage des racines). Le contenu de $\mathsf{AccField}_m$ est inclus dans cet ensemble si le mutateur m est en cours de marquage, puisque dans ce cas l'action 26 va placer dans *toMark* l'objet créé, qui contient le contenu de AccField.

$$\mathsf{ReachWh} \qquad \triangleq \mathsf{ReachMark}(\mathit{Pid}) \cup \mathit{toBlack} \cup \mathit{toTrace}$$

Les objets de ReachWh sont sûrs d'être examinés par le collecteur, quelles que soient les modifications faites au graphe mémoire. ReachWh contient donc au moins les objets blancs que le GC est sûr de tracer (ReachMark(Pid)) et ceux qu'il est déjà en train de tracer (*toTrace* et *toBlack*).

$$\mathsf{ReachFields} \qquad \triangleq \widehat{\mathit{fields}} \backslash \bigcup \mathsf{Kill}_{Pid}$$

ReachFields est l'ensemble des champs dont le GC va examiner le contenu. Les champs volatils n'en font pas partie car leur contenu est susceptible de changer avant que le GC l'examine.

$$\mathsf{ScanDone} \quad \triangleq \begin{cases} \emptyset & \textbf{if } \mathit{reset} \vee \mathit{dirty} \\ [0, \mathit{ptr}[& \textbf{otherwise} \end{cases}$$

ScanDone est la partie du tas que l'étape *Scan* a déjà parcourue et ne va pas reparcourir. Elle est vide si *dirty* (ou *reset*) est vrai car dans ce cas il faudra faire un nouveau parcours complet avant que l'étape *Scan* puisse se terminer. ScanDone est vide lorsque le GC n'est pas dans l'étape *Scan* car dans ce cas les invariants montrent que *reset* est toujours vrai.

$$\mathsf{Reach} \quad \triangleq \mathsf{ReachWh} \cup (\mathsf{Gr}\backslash\mathsf{ScanDone}) \cup \widehat{\mathit{cache}} \cup \mathsf{Hp}(\mathsf{ReachFields})$$

Reach est l'ensemble des objets que le GC va examiner. Les objets blancs et gris de Reach et les objets du cache constituent l'ensemble des objets "marqués" (selon la terminologie de la section 3.1). Quant aux autres objets de Reach, le GC va simplement constater leur noirceur et les ignorer.

$$\mathsf{Mrk} \quad \triangleq \mathsf{Tr}^*(\mathsf{Reach}) \cup (\mathsf{Val}\backslash\mathsf{Wh})$$

Mrk est l'ensemble des objets qui seront marqués (et donc considérés comme accessibles par la prochaine étape *Sweep*) si les mutateurs n'interfèrent pas avec le travail du collecteur. C'est l'ensemble des objets accessibles à partir de Reach par la relation de traçage, plus l'ensemble des objets valides qui sont noirs ou gris.

$$\mathsf{Q} \quad \triangleq \{p \in \mathit{Pid} \mid \mathit{status}_p = \mathit{Quick}\}$$

Q est l'ensemble des processus qui sont en cours de lancement.

$$\mathsf{L}_m \quad \triangleq \begin{cases} \{\mathit{child}_m\} & \textbf{if } \mathit{pc}_m = \mathit{Launch} \\ \emptyset & \textbf{otherwise} \end{cases}$$

L_m contient le processus que le mutateur m est en train de lancer, s'il existe.

$$\mathsf{N}_m \quad \triangleq \begin{cases} \{\mathit{new}_m\} & \textbf{if } \mathit{pc}_m \in \{\mathit{TestSweep}, \mathit{ClearNew}, \mathit{GrayNew}\} \\ \emptyset & \textbf{otherwise} \end{cases}$$

N_m contient éventuellement l'objet dont le mutateur m est en train de déterminer la couleur de création. L'objet est déjà créé, mais il n'a pas encore la bonne couleur.

6.2 Les invariants

Cette section présente quarante six prédicats sur les états du programme. Le but de la preuve est d'établir que ces prédicats sont vrais pour tous les états de tous les comportements possibles de l'algorithme. C'est pourquoi nous appelons ces prédicats des invariants. Chaque invariant dépend implicitement de l'état courant en faisant référence aux variables du programme et aux fonctions d'état définies dans la section précédente.

Certains des invariants découlent simplement du typage des variables. Il n'y a pas de notion intrinsèque de type en TLA : le fait que les opérations du programme sont bien typées est une propriété du programme que l'on prouve en TLA sans qu'il y ait besoin de mécanisme spécial. Nous avons donné les types des variables dans notre modèle, mais seulement pour aider la lecture : le typage des variables apparaît aussi dans les invariants (sauf pour les booléens).

6.2.1 Structure de la mémoire

Les quatre premiers invariants décrivent le contenu de la mémoire et des variables globales. Ce sont des invariants fondamentaux qui concernent toutes les actions du programme. On y trouve en particulier le chaînage du tas et le recensement des objets bleus.

$I_1 \triangleq heap \in \mathbf{array}\ [Addr]\ \mathbf{of}\ \ Words\ \wedge\ top \in \mathsf{A}^*(0)$

La première clause de cet invariant est une condition de typage : la mémoire est une fonction ($heap$) de $Addr$ dans $Words$ et elle devra le rester. En particulier, les opération d'écriture dans la mémoire, qui changent $heap$, devront toujours écrire une valeur de type $Words$ pour préserver cet invariant.

La deuxième clause est plus intéressante. Elle affirme que les en-têtes définissent un pavage du tas : en partant de l'objet situé à l'adresse 0 et en suivant la relation d'adjacence A, on arrive à l'adresse top, qui est la fin du tas.

$I_2 \triangleq alloc \oplus \bigoplus \mathsf{FreeList}_{Pid} = \mathsf{Bu}$

Cet invariant donne le recensement des objets bleus. L'ensemble Bu des objets bleus doit être égal au multi-ensemble qui est l'union de la liste libre globale, des listes libres locales et des objets provisoirement retirés des listes libres locales. Cela signifie que chaque objet bleu doit apparaître exactement une fois dans l'un de ces multi-ensembles. Chaque objet bleu est donc soit dans la liste libre globale soit sous la responsabilité d'un unique mutateur.

Cette propriété sert à prouver qu'il n'y a pas de fuite de mémoire, mais aussi qu'un objet ne peut pas être alloué par deux mutateurs simultanément.

$I_3 \triangleq free \oplus claim \subset \mathsf{Wh}$

Cet invariant affirme que tous les objets de $free$ et de $claim$ sont blancs, que l'intersection entre $free$ et $claim$ est vide, et qu'aucun objet n'apparaît plusieurs fois dans $free$ ou dans $claim$.

$I_4 \triangleq \mathsf{P}^*(\bigcup \mathsf{Acc}_{Pid}) \subset \mathsf{Val}\ \wedge\ \bigoplus \mathsf{Creating}_{Pid} \subset \mathsf{Obj} \backslash \mathsf{Fr}$

La première clause de cet invariant constitue le principal résultat de la preuve : les objets accessibles (par la relation P) à partir des pointeurs connus par les mutateurs

(Acc_m) sont des objets valides : des objets qui ne sont ni dans la liste libre, ni en cours de désallocation, ni en cours d'allocation.

La deuxième clause complète l'invariant en précisant que chaque objet en cours d'allocation est géré par un seul mutateur, et que ces objets ne peuvent pas être dans la liste libre ni en cours de désallocation.

6.2.2 Poignées de mains

Les deux invariants suivants décrivent le bon fonctionnement des poignées de mains, du point de vue du collecteur. Une partie de l'invariant I_{26} complète cette description.

$$I_5 \triangleq step \in Steps \land phase \in \{Async, Sync_1, Sync_2\}$$

I_5 est un invariant de typage qui précise que la variable *step* est bien typée et que la variable *phase* ne peut pas être *Dead*, *Quick* ni *Avail*, qui ne sont significatifs que pour les $status_m$.

$$\begin{aligned} I_6 \triangleq\ & \lor\ status_C = phase = Async \\ & \lor\ status_C = Sync_1 \land phase \neq Sync_2 \land step = Clear \\ & \lor\ status_C = Sync_2 \land phase \neq Async \land step = Mark \\ & \lor\ status_C = Async \land phase = Sync_2 \land step = Mark \end{aligned}$$

Cet invariant détaille tous les cas possibles en ce qui concerne les relations entre les poignées de mains et les étapes du collecteur (cf. figure 4.13) :

- Si $status_C = phase = Async$, on se trouve entre la troisième poignée de mains et la première du cycle suivant.
- Si $status_C = Sync_1$, alors $phase = Async$ (pendant la première poignée de mains) ou $phase = Sync_1$ (entre la première et la deuxième). Dans les deux cas, l'étape courante est *Clear*.
- De même, si $status_C = Sync_2$ (entre le début de la deuxième poignée de mains et celui de la troisième), alors l'étape est *Mark*.
- Enfin, pendant la troisième poignée de mains l'étape courante est *Mark*.

6.2.3 Désallocation

Ces invariants concernent particulièrement les actions du collecteur qui s'occupent de désallouer les objets morts.

$$\begin{aligned} I_7 \triangleq\ & \land\ limit \in \mathsf{Obj} \cup \{top\} \\ & \land\ ptr \in \mathsf{Obj} \cup \{top\} \land ptr \leq limit \\ & \land\ sublimit \in Addr \land sublimit \leq limit \end{aligned}$$

Cet invariant dit que :

- *limit* et *ptr* pointent vers des objets du tas ou bien juste après le dernier objet du tas,
- *sublimit* est une adresse quelconque, et
- *ptr* et *sublimit* se trouvent avant *limit*.

Il servira à montrer que l'étape *Sweep* se déroule correctement. Les propriétés de *limit* et *ptr* servent aussi à établir le bon fonctionnement des parcours du tas effectués par les étapes *Clear* et *Scan*.

$$I_8 \triangleq claim \subset \mathsf{Wh} \cap [0, ptr[\ \wedge\ (step \in \{Mark, Scan\} \ \Rightarrow\ claim = \emptyset)$$

Cet invariant affirme que les objets que le collecteur a décidé de désallouer sont tous des objets blancs qui viennent d'être balayés et qu'aucun objet ne se trouve plusieurs fois dans *claim*. De plus, *claim* doit être vide pendant les étapes *Mark* et *Scan*.

$$I_9 \triangleq toWhite \subset (\mathsf{Obj} \backslash \mathsf{Fr}) \cap [0, ptr[\ \wedge\ (step \in \{Mark, Scan\} \ \Rightarrow\ toWhite = \emptyset)$$

Cet invariant spécifie que l'ensemble des objets que le collecteur a décidé de blanchir ne contient que des objets valides (c'est-à-dire des nœuds du graphe mémoire) et des objets en cours d'allocation (car $\mathsf{Obj} \backslash \mathsf{Fr}$ est égal à Val plus les $\mathsf{Creating}_m$ qui sont dans Obj). En particulier, le collecteur ne pourra pas blanchir (et donc désallouer une deuxième fois) les objets qui sont déjà bleus. De plus, les objets à blanchir sont dans la partie $[0, ptr[$ du tas qui a déjà été balayée, et ils n'existent que pendant les étapes *Sweep* et *Clear*.

6.2.4 l'étape *Sweep*

Ces invariants concernent le fonctionnement du système lors de l'étape *Sweep* du GC.

$$\begin{aligned} I_{10} \triangleq\ & \vee\ swept = -\infty \ \wedge\ (step = Clear \ \Rightarrow\ phase \neq Async) \\ & \vee\ swept \in Addr \ \wedge\ step = Sweep \ \wedge\ swept \leq ptr \\ & \vee\ swept = +\infty \ \wedge\ step \neq Scan \\ & \qquad\qquad\qquad\ \wedge\ (step = Mark \ \Rightarrow\ status_C \neq Async) \\ & \qquad\qquad\qquad\ \wedge\ (step = Sweep \ \Rightarrow\ ptr = limit) \end{aligned}$$

I_{10} donne la relation entre les valeurs de *swept* et le cycle du GC. La variable *swept* vaut $-\infty$ pendant l'étape *Scan*. Elle parcourt les adresses du tas lors de l'étape *Sweep*, et passe à $+\infty$ juste avant la fin de cette étape. Elle reste ensuite à $+\infty$ jusqu'à ce que le cycle de GC soit fini, et que tous les mutateurs le sachent, donc au moins jusqu'à la fin de la première poignée de mains. Elle doit repasser à $-\infty$ avant que les mutateurs repassent en mode *Async* (car l'action 22 du mutateur utilise la valeur de *swept* pour prendre sa décision si $status_m = Async$), donc avant le début de la troisième poignée de mains.

On peut donc avoir :

- $swept = -\infty$ pendant toutes les étapes. Pendant l'étape *Clear*, seulement après la fin de la première poignée de mains.
- $swept \in Addr$ seulement pendant l'étape *Sweep*. Dans ce cas, *swept* ne peut pas dépasser *ptr*.
- $swept = +\infty$ sauf pendant l'étape *Scan*. Pendant l'étape *Mark*, seulement avant le début de la troisième poignée de main. Pendant l'étape *Sweep* seulement si $ptr = limit$ (l'étape *Sweep* est alors sur le point de se terminer).

Cet invariant servira a montrer la validité des décisions prises par l'action 22 du mutateur. Par exemple, le test $swept > new_m$ ne peut pas être vrai pendant l'étape *Scan*.

$I_{11} \triangleq step = Sweep \;\Rightarrow\; \mathsf{Bk} \subset toWhite \cup \bigcup \mathsf{N}_{Pid} \cup [ptr, top[$

Cet invariant affirme que, pendant l'étape *Sweep*, tous les objets noirs sont dans l'ensemble *toWhite* des objets que le collecteur a décidé de blanchir, ou dans la partie $[ptr, top[$ du tas qui reste à balayer, ou parmi les objets en cours d'allocation qui sont provisoirement noirs en attendant que les mutateurs leur donnent leur couleur d'allocation. Cet invariant servira à établir I_{14} au moment du passage de l'étape *Sweep* à l'étape *Clear*.

$I_{12} \triangleq step = Sweep \;\Rightarrow\; free \cap [ptr, sublimit[= \emptyset$

Pendant l'étape *Sweep*, le GC cherche les objets blancs compris entre *ptr* et *sublimit* pour les placer dans *claim*. Ceux-ci ne doivent pas être déjà dans *free* (a cause de l'invariant I_3).

$I_{13} \triangleq step = Sweep \;\Rightarrow\; (toWhite \cup \mathsf{Wh} \cup \mathsf{Bu}) \cap]swept, ptr[= \emptyset$

L'intervalle ouvert $]swept, ptr[$ ne contient en principe aucun objet (car *ptr* pointe sur le même objet que *swept* ou sur le suivant), sauf si un ou des mutateurs viennent de découper (par l'action 21) un objet bleu pour créer de nouveaux objets. Ces nouveaux objets ne peuvent être ni bleus, ni blancs, ni éléments de *toWhite*.

Cet invariant sert à établir l'invariant 37 au moment où on exécute l'action 21 car old_m est dans Bu lors de cette action. On ajoute *toWhite* et Wh à Bu dans I_{13} pour établir I_{13} lors du passage des objets de Wh à Bu et de *toWhite* à Wh (actions 2 et 40).

6.2.5 L'étape *Clear*

Les deux invariants suivants concernent l'étape *Clear* du GC.

$I_{14} \triangleq step = Clear \;\Rightarrow\; \mathsf{Bk} \subset toWhite \cup \bigcup \mathsf{N}_{Pid} \cup [ptr, limit[$

Lors de l'étape *Clear*, les objets noirs sont dans *toWhite*, dans la partie $[ptr, limit[$ du tas qui va être balayée ou ce sont des objets en cours de création auxquels les mutateurs vont donner leur couleur définitive (les éléments des N_m). Cet invariant sert avec I_{15} qu'il n'y a plus d'objets noirs dans le tas lorsque le cycle suivant démarre (à part les objets en cours de création).

$$I_{15} \triangleq step = Clear \ \wedge \ status_C = Sync_1 \ \Rightarrow \ ptr = limit \ \wedge \ toWhite = \emptyset$$

Lorsque l'action 42 est activée, donc quand l'étape *Mark* est sur le point de commencer, il n'y a plus d'objets dans *toWhite* ni dans l'intervalle $[ptr, limit[$.

6.2.6 La procédure *Trace*

Les six invariants suivants décrivent le bon fonctionnement de la procédure *Trace* du GC (actions 50 à 54).

$$I_{16} \triangleq rover \in \mathsf{Obj} \cup \{top\} \ \wedge \ (step \notin \{Mark, Scan\} \ \Rightarrow \ rover = 0)$$

La variable *rover* pointe toujours sur un objet ou sur la fin du tas. Elle reste nulle pendant les étapes *Sweep* et *Clear*. Cet invariant sert à prouver que l'action 49 ne met que des adresses d'objets (et pas des adresses arbitraires) dans *toBlack*.

$$I_{17} \triangleq toBlack \subset \mathsf{Val} \backslash \mathsf{Bk} \ \wedge \ (step \notin \{Mark, Scan\} \ \Rightarrow \ toBlack = \emptyset)$$

L'ensemble *toBlack* des objets que le collecteur a décidé de noircir ne contient que des objets valides qui ne sont pas noirs. Il est vide pendant les étapes *Sweep* et *Clear*, donc l'action 50 n'est jamais activée pendant ces étapes.

$$I_{18} \triangleq toTrace \subset \mathsf{Val} \ \wedge \ (step \notin \{Mark, Scan\} \ \Rightarrow \ toTrace = \emptyset)$$

L'ensemble *toTrace* des objets que le collecteur va examiner (il sait qu'ils sont vivants mais il n'a pas encore regardé leur couleur) ne contient que des objets valides. Il est vide pendant les étapes *Sweep* et *Clear*.

$$I_{19} \triangleq cache \in \textbf{multiset of } \mathsf{Val} \backslash \mathsf{Wh} \ \wedge \ (step \notin \{Mark, Scan\} \ \Rightarrow \ cache = \emptyset)$$

Le cache ne contient que des objets valides qui ne sont pas blancs. En principe, les objets du cache sont noirs, mais il peut arriver qu'un mutateur prenne la décision de griser un objet (l'action 29 active l'action 30), et que cet objet soit grisé puis noirci et placé dans le cache avant que le mutateur le grise (l'action 30 se déclenche). L'objet est alors gris et dans le cache.

$$I_{20} \triangleq fields \in \textbf{multiset of } \mathsf{C}(\mathsf{Val} \backslash \mathsf{Wh}) \ \wedge \ (step \notin \{Mark, Scan\} \ \Rightarrow \ fields = \emptyset)$$

Cet invariant est le reflet du précédent pour les objets qui ont été traités par l'action 52 : *fields* ne contient que des adresses de champs d'objets valides non blancs.

$$
\begin{aligned}
I_{21} \triangleq\ & \wedge\ (step \in \{Mark, Scan\} \\
& \quad \Rightarrow\ \wedge\ \mathsf{Hp}(\bigcup \mathsf{Kill}_{Pid}) \cup \mathsf{P}(\mathsf{Mrk}) \subset \mathsf{Mrk} \subset \mathsf{Val} \\
& \quad\quad \wedge\ (status_C \neq Async\ \Rightarrow\ \mathsf{Wh} \cap \mathsf{Hp}(\mathsf{FDone} \cup \bigcup \mathsf{Kill}_{Pid}) \subset \mathsf{ReachWh})) \\
& \wedge\ (step \notin \{Mark, Scan\}\ \Rightarrow\ \mathsf{P}^*(\mathsf{Gr} \cap \mathsf{Val}) \subset \mathsf{Val})
\end{aligned}
$$

I_{21} est l'un des invariants les plus importants. Il est composé de deux cas. Lors des étapes *Mark* et *Scan*, le collecteur est en train de parcourir le graphe mémoire pour trouver les objets accessibles. Dans ce cas, l'invariant comporte quatre clauses :

- L'ensemble des objets pointés par les champs volatils est inclus dans Mrk, ce qui signifie que ces objets seront quand même marqués par le collecteur, même si les pointeurs volatils disparaissent. Cette clause affirme donc que la procédure *UpdateProc* fonctionne correctement : elle s'assure que l'ancienne valeur sera tracée avant de l'effacer par affectation.

- L'ensemble Mrk des objets qui seront marqués est clos par la relation P d'accessibilité. Cela signifie que les mutateurs (qui doivent suivre P pour accéder à de nouveaux objets) ne pourront pas sortir de Mrk.

- L'ensemble Mrk est inclus dans l'ensemble Val des objets valides. Cela assure que le collecteur (qui parcourt le graphe mémoire en suivant les pointeurs contenus dans les objets de Mrk) n'essayera pas de lire des pointeurs invalides.

- Entre le début de la deuxième et le début de la troisième poignée de mains (quand $status_C \neq Async$), les objets blancs pointés par les champs que le GC ne verra pas (les champs des objets parcourus et les champs volatils) seront quand même tracés par le GC quelles que soient les écritures effectuées. Cette clause sert à établir la correction des écritures effectuées par les mutateurs qui n'ont pas encore répondu à la deuxième poignée de mains.

Lors des étapes *Sweep* et *Clear*, cet invariant assure simplement que les objets gris valides et leurs descendants ne pointent que sur des objets valides. Cette clause permettra d'assurer que le collecteur ne suivra que des pointeurs valides à partir de ces objets lorsqu'il les détectera au cours de la prochaine étape *Scan*.

6.2.7 L'étape *Scan*

Ces quatre invariants décrivent le bon fonctionnement de l'étape *Scan* et la cohérence de la variable *scanned*.

$I_{22} \triangleq scanned \in Addr \cup \{-\infty\}$

I_{22} est un simple invariant de typage.

$I_{23} \triangleq \neg\, reset\ \Rightarrow\ step = Scan\ \wedge\ scanned \leq ptr$

Cet invariant affirme que *reset* doit toujours être vrai hors de l'étape *Scan* et que *scanned* ne doit pas dépasser *ptr* si *reset* est faux. La valeur de *scanned* n'a pas d'importance si *reset* est vrai car dans ce cas le collecteur va remettre *scanned* à $-\infty$.

$I_{24} \triangleq step = Scan \;\Rightarrow\; \mathsf{Mrk}\backslash\mathsf{Done} \subset [0, limit[$

Pendant l'étape *Scan*, les objets que le collecteur devra trouver (les objets à marquer qui ne sont pas encore noirs) sont tous en deçà de *limit*. Comme le parcours de recherche des objets gris s'arrête à *limit*, cet invariant sert à prouver que le collecteur va bien trouver tous les objets gris.

$$I_{25} \triangleq step = Scan \wedge \neg\, dirty \wedge \neg\, reset \wedge \mathsf{Mrk} \cap \mathsf{Wh} \neq \emptyset \\ \quad \Rightarrow\;]scanned, ptr[\cap \mathsf{Obj} \subset \mathsf{Done}$$

Pendant l'étape *Scan*, si on est réellement pendant un parcours ($dirty \vee reset$ indique qu'on va démarrer un nouveau parcours, $\mathsf{Mrk} \cap \mathsf{Wh} = \emptyset$ indique que le dernier parcours est fini), alors *scanned* est "approximativement égal" à *ptr*. Les objets de l'intervalle $]scanned, ptr[$ sont donc forcément des objets qui viennent d'être créés par un ou des mutateurs. Cet invariant affirme que les mutateurs prennent la bonne décision quant à la couleur des objets créés pendant l'étape *Scan*.

6.2.8 Gestion des processus

$I_{26} \triangleq \forall m \in Pid, status_m \in \{status_C, phase, Dead, Avail, Quick\}$

Cet invariant dit que chaque processus se trouve soit dans les divers stades de mort et de résurrection (*Dead*, *Avail* ou *Quick*), soit entre $status_C$ et *phase*. Hors des poignées de mains, on a $status_C = phase$, donc les mutateurs vivants doivent avoir $status_m = status_C$. Pendant les poignées de mains, $status_m$ ne peut pas prendre la troisième valeur (parmi $Sync_1$, $Sync_2$ et *Async*) : il doit se trouver entre $status_C$ et *phase*.

$I_{27} \triangleq \bigoplus \mathsf{L}_{Pid} = \mathsf{Q}$

Cet invariant affirme simplement que l'ensemble Q des processus en cours de lancement est égal au multi-ensemble des fils des processus qui sont en train d'en lancer d'autres. Autrement dit, chaque processus fils a un unique père.

$I_{28} \triangleq \forall m \in Pid, args_m \in \textbf{multiset of } Addr$

Cet invariant donne le type des variables $args_m$, qui servent à communiquer des racines initiales aux nouveaux processus.

6.2.9 La procédure *Cooperate*

$I_{29} \triangleq \forall m \in Pid, answering_m \vee marking_m \Rightarrow pc_m \neq Halt \wedge status_m \neq status_C$

Pendant l'exécution de *Cooperate* (quand $answering_m \vee marking_m$ est vrai), le mutateur m ne peut pas être à l'arrêt ($pc_m \neq Halt$) et une poignée de mains doit être en cours à laquelle il n'a pas encore répondu ($status_m \neq status_C$).

$$I_{30} \triangleq \forall m \in Pid, \begin{array}{l} \wedge\ (marking_m \Rightarrow status_C = Async) \\ \wedge\ (status_m = Async \Rightarrow toMark_m = \emptyset) \end{array}$$

Le marquage des racines se fait au cours de la troisième poignée de mains (I_{29}) nous dit qu'une poignée de mains est en cours et $status_C = Async$ précise que c'est la troisième). Après la troisième poignée de mains, et jusqu'au prochain cycle de GC ($status_m = Async$), $toMark_m$ doit être vide. En effet, le marquage est fait, donc il est trop tard pour ajouter des objets à l'ensemble des objets à marquer.

$$I_{31} \triangleq \forall m \in Pid, (status_m = Async \vee marking_m) \wedge step \in \{Mark, Scan\} \\ \Rightarrow \mathsf{Acc}_m \subset \mathsf{Mrk}$$

Pendant le parcours du graphe mémoire par le GC ($step \in \{Mark, Scan\}$), à partir du moment où le mutateur m commence à marquer ses racines ($marking_m \vee status_m = Async$), l'ensemble Acc_m des racines étendues de m (tous les objets auxquels m peut accéder directement) est inclus dans l'ensemble Mrk des objets qui seront marqués par le GC. Comme Mrk est clos par la relation P (I_{21}), on pourra donc montrer que l'ensemble $\mathsf{P}^*(\mathsf{Acc}_m)$ des objets auxquels m peut accéder directement ou indirectement est inclus dans l'ensemble des objets que le collecteur considérera comme vivants. Autrement dit, cet invariant sert à montrer que le GC ne désalloue pas d'objets auxquels m pourrait accéder.

6.2.10 Processus morts

$I_{32} \triangleq \forall m \in Pid, pc_m \in Labels \wedge (status_m \in \{Dead, Avail, Quick\} \Rightarrow pc_m = Halt)$

$I_{33} \triangleq \forall m \in Pid, pc_m = Halt \Rightarrow roots_m = toMark_m = pool_m = \emptyset$

Les processus inactifs ou en cours de lancement sont forcément dans l'état *Halt*, et ils n'ont pas de racines, pas d'objets à marquer et pas de liste libre locale. Ces deux invariants prouvent donc qu'aucune action n'est activée pour ces processus, sauf éventuellement l'action 5 (démarrage du processus).

$I_{34} \triangleq \forall m \in Pid, pc_m \neq Halt \vee status_m \in \{Dead, Avail\} \Rightarrow args_m = \emptyset$

La variable $args_m$ est vide sauf pendant les étapes de lancement du processus m. Cet invariant nous permet de prouver que le mutateur m ne peut pas "oublier" de pointeurs dans $args_m$ pour ensuite les retrouver après que le GC a désalloué les objets correspondants.

6.2.11 Création et remplissage de nouveaux objets

$$I_{35} \triangleq \forall m \in Pid,\ \begin{array}{l} \wedge\ (pc_m \in CreateProc\ \Rightarrow\ toFill_m \subset \mathsf{Cf}(new_m)\ \wedge\ old_m \in Addr) \\ \wedge\ (pc_m \notin CreateProc\ \Rightarrow\ toFill_m = \emptyset) \end{array}$$

L'ensemble $toFill_m$ des champs restant à remplir n'existe que pendant la procédure de création, et il ne doit contenir que des champs de l'objet new_m en cours de création. La clause $old_m \in Addr$ est une simple propriété de typage.

$$I_{36} \triangleq \forall m \in Pid, pc_m = Split\ \Rightarrow\ \begin{array}{l} \wedge\ new_m \in \mathsf{Cf}(old_m)\ \wedge\ \mathsf{Af}(new_m) = \mathsf{Af}(old_m) \\ \wedge\ heap[new_m] \in Headers \\ \wedge\ heap[new_m].\, color = Black \end{array}$$

Si le mutateur m est sur le point de découper un objet (old_m), alors le nouvel objet qui va être créé a déjà un en-tête valide ($heap[new_m] \in Headers$), il est noir, il est à l'intérieur de l'objet découpé ($new_m \in \mathsf{Cf}(old_m)$) et il se trouve à la fin de cet objet ($\mathsf{Af}(new_m) = \mathsf{Af}(old_m)$). Cet invariant nous permettra de montrer que l'action 21 préserve le pavage du tas, bien qu'elle change la relation d'adjacence.

$$I_{37} \triangleq \forall m \in Pid,\ \begin{array}{l} \wedge\ pc_m = TestSweep\ \wedge\ step = Sweep \\ \wedge\ swept < old_m\ \wedge\ new_m \in [swept, ptr[\\ \Rightarrow\ new_m \in \mathsf{Wh} \cup toWhite \end{array}$$

Si le mutateur m est sur le point d'exécuter l'action 22 et de prendre la troisième branche dans le test effectué par cette action (le mutateur considère que l'objet va être balayé et vu noir par le collecteur, donc blanchi) et si l'objet a déjà été balayé ($new_m \in [swept, ptr[$), alors l'objet va être blanchi (il est dans $toWhite$) ou a déjà été blanchi (il est dans Wh). Cet invariant nous permet de prouver que la troisième branche de l'action 22 ne peut pas laisser passer un objet noir au travers du balayage.

$$I_{38} \triangleq \forall m \in Pid, step \in \{Mark, Scan\}\ \vee\ status_m \neq Async \\ \Rightarrow\ \mathsf{Creating}_m \subset \mathsf{Wh} \cup \mathsf{Bk}\ \wedge\ pc_m \neq GrayNew$$

À partir du moment où m a répondu à la première poignée de mains et jusqu'à la fin de l'étape $Scan$, l'objet en cours de création ne peut pas être gris (et il n'est pas sur le point d'être grisé). En effet, les champs de cet objet ne sont pas encore initialisés, donc il ne faut pas que le collecteur essaye de les lire. Or pendant les étapes $Mark$ et $Scan$, le collecteur cherche les objets gris pour les tracer. Cet invariant sert donc à prouver que le collecteur ne va pas essayer de suivre des pointeurs invalides.

$$I_{39} \triangleq \forall m \in Pid, step \in \{Mark, Scan\}\ \wedge\ status_m = Async \\ \Rightarrow\ \mathsf{Creating}_m \subset \mathsf{Bk}\ \wedge\ pc_m \neq ClearNew$$

À partir du moment où le mutateur m a répondu à la troisième poignée de mains et jusqu'à la fin de l'étape $Scan$, l'objet en cours de création ne peut être que noir, et le mutateur m n'est pas sur le point de le colorier en blanc. Cet invariant est utilisé pour prouver qu'à la fin de l'étape $Scan$ il n'y a pas d'objets vivants blancs qui viennent d'être créé par un mutateur.

$I_{40} \triangleq \forall m \in Pid, pc_m = ClearNew \wedge step = Sweep \Rightarrow new_m < swept$

Si le mutateur a pris la décision de colorier en blanc l'objet créé ($pc_m = ClearNew$) pendant l'étape de balayage, alors l'objet se trouve dans la partie du tas qui a déjà été balayée. Cet invariant sert à prouver que la première branche de l'action 22 est correcte.

$I_{41} \triangleq \forall m \in Pid, pc_m = Fill \wedge status_m \neq Async \Rightarrow new_m \in \mathsf{Wh}$

Si le mutateur m donne à un nouvel objet sa couleur définitive avant de répondre à la troisième poignée de mains, alors cet objet doit être blanc. Le mutateur peut placer dans cet objet certaines de ses racines et omettre de tracer ces racines (action 25). Il faut donc être sûr que cet objet sera tracé par le collecteur, donc il ne doit pas être noir. Il ne doit pas non plus être gris car le collecteur pourrait alors essayer de le tracer et donc de lire les champs non initialisés. Il ne peut pas être bleu car il a déjà pris sa couleur d'allocation et le mutateur s'apprête à le placer dans le graphe mémoire.

6.2.12 Modification du graphe mémoire

Nos cinq derniers invariants décrivent le bon fonctionnement des actions de *UpdateProc*.

$I_{42} \triangleq \forall m \in Pid, pc_m \in UpdateProc \Rightarrow field_m \in \mathsf{C}(\mathsf{Val})$

L'objet dans lequel le mutateur m va faire son écriture doit rester valide pendant toute la durée de la procédure *UpdateProc*.

$I_{43} \triangleq \forall m \in Pid, pc_m \in UpdateProc \setminus \{Store\} \Rightarrow status_m = Async$

Les actions 29 à 32 ne seront pas activées pendant les poignées de mains, ou plus précisément entre le moment où le mutateur répond à la première poignée de mains et le moment où il répond à la troisième.

$$
\begin{aligned}
I_{44} \triangleq \forall m \in Pid, pc_m = GrayOld \wedge step = Scan & \\
\Rightarrow \wedge\ & old_m < limit \\
\wedge\ & (old_m \in]scanned, ptr[\Rightarrow dirty \vee reset \vee \mathsf{Mrk} \cap \mathsf{Wh} = \emptyset)
\end{aligned}
$$

Si le mutateur est sur le point de griser old_m (l'ancienne valeur supposée de l'affectation), et si le collecteur est dans son étape *Scan*, alors old_m se trouve dans la partie du tas qui est parcourue par l'étape *Scan* et si old_m a été créé récemment (il se trouve entre *scanned* et *ptr*), alors le GC fera un nouveau parcours ou bien il a déjà marqué tout ce qui doit être marqué (y compris old_m). Cet invariant sert à établir I_{24} et I_{25} en prouvant que les actions de modification ne peuvent pas changer intempestivement la couleur définitive des objets nouvellement créés.

$$
\begin{aligned}
I_{45} \triangleq \forall m \in Pid, pc_m \in \{TestScan, SetDirty\} \wedge step \in \{Mark, Scan\} & \\
\Rightarrow old_m \in \mathsf{Bk} \cup \mathsf{Gr} &
\end{aligned}
$$

Entre le moment où le mutateur m grise old_m ou constate qu'il est gris et le moment où il positionne *dirty*, si l'étape courante est *Mark* ou *Scan* alors old_m est gris ou noir (il a pu être noirci par le collecteur après avoir été grisé par le mutateur). Cet invariant sert à établir I_{31} puisque old_m est un élément de Acc_m et le pointeur contenu dans $heap[field_m]$ qui pointe vers old_m va devenir volatil, donc old_m doit être dans la partie $\mathsf{Val}\backslash\mathsf{Wh}$ de Mrk.

$$\begin{aligned} I_{46} \triangleq \forall m \in Pid, pc_m = Store \ \wedge \ status_m \neq Async \ \wedge \neg\, marking_m \\ \Rightarrow \ new_m \in \widehat{toMark}_m \cup (\mathsf{Val}\backslash\mathsf{Wh}) \end{aligned}$$

Entre la réponse de m à la première poignée de mains et le moment où il commence à marquer, la nouvelle valeur de l'affectation est un objet qui sera (ou est déjà) marqué. C'est la condition qui garantit que le scénario de la figure 4.9 ne peut pas se produire.

6.3 Plan de la preuve

Cette section donne un aperçu de la preuve des invariants exposés dans la section précédente. Nous ne donnerons pas la preuve complète car elle est très volumineuse.

On définit l'invariant

$$I = \bigwedge_{i=1}^{46} I_i$$

et, en notant a_j l'action numéro j, nous définissons l'action a qui représente tout le programme par :

$$a = \bigvee_{j=1}^{63} a_j$$

Pour prouver que l'invariant I est vrai pour tous les états de tous les comportements possibles de notre programme, nous prouvons d'une part que les conditions initiales impliquent I, c'est-à-dire que tout état qui vérifie les conditions initiales vérifie l'invariant ; d'autre part que l'action a préserve I, c'est-à-dire que pour toute paire d'états (s,t) on a : $I(s) \ \wedge \ a(s,t) \ \Rightarrow \ I(t)$. En TLA, on note cette propriété avec l'opérateur "prime" :

$$I \ \wedge \ a \ \Rightarrow \ I'$$

Cela revient à prouver :

$$\forall i \in [1,46], \forall j \in [1,63], I \ \wedge \ a_j \ \Rightarrow \ I'_i$$

Clauses triviales

Il faut d'abord prouver que les conditions initiales impliquent l'invariant, ce qui est très facile.

Il nous faut ensuite prouver 2898 propositions de la forme $I \ \wedge \ a_j \ \Rightarrow \ I'_i$ (que nous noterons P_{ij}).

I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
a_{1}	•	•	×	•	·	·	•	×	•	·	•	·	•	×	·	•	×	×	×	×	•	·	·	×	•	·	·	·	·	·	×	·	·	·	•	•	×	×	×	·	×	×	·	×	×	×
a_{2}	•	•	•	×	·	·	×	•	×	·	×	•	•	×	·	×	×	×	×	×	•	·	·	×	•	·	·	·	·	·	×	·	·	·	×	•	•	•	×	·	•	×	·	•	×	×
a_{3}	·	·	•	•	·	·	·	·	•	·	·	•	·	·	·	·	•	•	×	•	•	·	·	×	×	·	·	·	·	·	×	·	·	·	·	•	·	·	·	·	·	•	·	×	·	×
a_{4}	•	•	•	•	·	·	×	•	•	·	×	·	•	×	·	×	•	•	×	•	•	·	·	×	•	·	·	·	·	·	×	·	·	·	×	•	•	•	×	·	•	•	·	•	×	×
a_{5}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	×	×	·	×	•	•	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{6}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	×	×	•	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{7}	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	×	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·
a_{8}	·	×	·	•	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	×	×	•	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{9}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	×	×	•	×	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{10}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{11}	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	•	•	•	×	·	•	•	•	•	•	•	·	·	·	•	•	·	•	·	•	•	·	•
a_{12}	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	×	×	•	×	·	•	•	•	•	·	•	·	·	·	•	•	·	•	·	•	×	·	•
a_{13}	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	×	×	·	·	·	·	•	•	·	•	·	·	·	·	·	·	·	·	·	·	×	·	•
a_{14}	•	×	•	•	·	·	×	•	×	·	•	·	•	•	·	×	•	×	•	•	•	·	·	•	•	·	·	·	·	•	•	·	•	·	×	•	•	•	•	·	•	×	·	•	•	•
a_{15}	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	×	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{16}	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{17}	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{18}	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{19}	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{20}	•	•	×	•	·	·	×	×	•	·	•	·	•	•	·	×	•	×	×	×	•	·	·	×	×	·	×	·	•	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{21}	•	×	×	•	·	·	•	×	•	·	•	·	×	•	·	•	•	×	×	×	•	·	·	•	•	·	×	·	•	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{22}	·	×	·	×	·	·	·	·	·	·	•	·	·	•	·	·	×	×	×	×	•	·	·	×	×	·	×	·	•	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{23}	•	×	•	×	·	·	×	•	×	·	•	·	•	•	·	×	•	×	×	×	•	·	·	•	•	·	×	·	•	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{24}	•	×	•	×	·	·	×	•	×	·	•	·	•	•	·	×	•	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{25}	•	×	×	×	·	·	×	×	×	·	×	·	×	×	·	×	×	×	×	×	•	·	·	×	×	·	·	·	·	•	×	·	•	·	•	•	×	×	×	·	×	×	·	×	×	•
a_{26}	·	×	·	•	·	·	·	·	·	·	×	·	·	×	·	·	•	•	•	•	•	·	·	•	•	·	×	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{27}	·	×	·	•	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	•	•	·	×	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{28}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{29}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{30}	•	×	•	×	·	·	×	•	×	·	•	·	•	•	·	×	•	×	•	•	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{31}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{32}	·	×	·	×	·	·	·	·	·	·	×	·	·	×	·	·	×	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{33}	•	×	×	•	·	·	×	×	×	·	×	·	×	×	·	×	×	×	×	×	•	·	·	•	•	·	×	·	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
a_{34}	·	·	•	×	·	·	•	•	•	•	•	•	•	•	•	·	×	×	×	×	×	·	•	×	•	·	·	·	·	·	×	·	·	·	·	×	•	·	·	·	·	×	·	•	·	×
a_{35}	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·
a_{36}	·	·	•	×	·	·	•	·	×	·	·	•	·	·	·	·	×	×	×	×	×	·	·	×	×	·	·	·	·	·	×	·	·	·	·	×	·	·	·	·	·	×	·	×	·	×
a_{37}	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·
a_{38}	·	·	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	·	•	•	•	·	·	·	·	·	•	·	·	·	·	×	•	•	•	•	·	×	·	•	•	×
a_{39}	·	·	·	×	·	·	•	•	•	•	•	•	•	•	•	·	×	×	×	×	×	·	•	×	•	·	·	·	·	·	×	·	·	·	·	×	•	·	·	·	·	×	·	•	·	×
a_{40}	•	×	•	×	·	·	×	•	•	·	•	·	•	•	•	×	•	×	•	•	•	·	·	•	•	·	·	·	·	·	•	·	·	·	×	•	•	•	•	·	•	×	·	•	•	•
a_{41}	·	·	·	·	·	•	·	·	·	•	·	·	·	·	•	·	·	·	·	·	•	·	·	·	·	•	·	·	•	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{42}	·	·	·	×	•	•	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	·	•	•	•	•	·	·	•	•	•	·	·	·	·	×	•	•	•	•	·	×	·	•	•	×
a_{43}	•	×	•	•	·	·	•	•	•	·	×	·	•	×	·	•	×	×	×	×	•	·	·	×	•	·	·	·	·	·	×	·	·	·	•	•	•	•	×	·	•	×	·	•	×	×
a_{44}	·	·	•	×	·	·	·	•	×	·	·	•	·	·	·	·	×	×	×	×	×	·	·	×	×	·	·	·	·	·	×	·	·	·	·	×	·	·	·	·	·	×	·	×	·	×
a_{45}	·	·	·	·	•	•	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{46}	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	×	·	·	×	×	•	×	·	•	•	•	•	·	•	·	·	·	•	•	·	•	·	•	×	·	•
a_{47}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{48}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{49}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	·	•	·	·	•	•	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·
a_{50}	•	×	•	×	·	·	×	•	×	·	•	·	•	•	·	×	•	×	•	•	•	·	·	×	•	·	·	·	·	·	×	·	·	·	×	•	•	•	•	·	•	×	·	•	•	•
a_{51}	•	×	•	×	·	·	×	•	×	·	•	·	•	•	·	×	•	×	•	×	•	·	•	•	•	·	·	·	·	·	•	·	·	·	×	•	•	•	•	·	•	×	·	•	•	×
a_{52}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	•	×	·	·	•	•	·	·	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	×	·	·
a_{53}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	•	•	·	·	•	•	·	·	·	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·
a_{54}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	•	·	·	•	·	·	×	×	·	·	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	×	·	·
a_{55}	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·	•	·	·	·	·	·	·
a_{56}	·	·	·	·	·	•	·	·	·	•	·	·	·	·	•	·	·	·	·	·	•	·	·	·	·	•	·	·	•	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
a_{57}	·	·	·	×	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	·	•	•	•	·	·	·	·	·	•	·	·	·	·	×	•	•	•	•	·	×	·	•	•	×
a_{58}	·	·	·	×	·	·	•	•	•	•	•	•	•	•	•	·	×	×	×	×	×	•	•	×	•	·	·	·	·	·	×	·	·	·	·	×	•	·	·	·	·	×	·	•	·	×
a_{59}	·	·	·	×	·	·	•	•	•	•	•	•	•	•	•	·	×	×	×	×	×	·	•	×	•	·	·	·	·	·	×	·	·	·	·	×	•	·	·	·	·	×	·	•	·	×
a_{60}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	×	·	•	×	•	·	·	·	·	·	×	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·
a_{61}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	•	·	•	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	•	·	·
a_{62}	·	·	·	×	·	·	•	•	•	•	•	•	•	•	•	·	•	×	×	×	•	·	•	×	•	·	·	·	·	·	×	·	·	·	·	×	•	·	·	·	·	×	·	•	·	×
a_{63}	·	·	·	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	·	•	•	•	·	·	·	·	·	•	·	·	·	·	•	•	•	•	•	·	•	·	•	•	×

Figure 6.1: Les 2898 cas de la preuve

Si a_j préserve toutes les variables qui interviennent dans l'expression de I_i (et des fonctions d'état dont I_i dépend), alors P_{ij} est trivialement vraie, puisque dans ce cas $I_i \wedge a_j \Rightarrow I_i'$. Les 1640 P_{ij} qui ont cette forme sont notées par un petit point ($\cdot$) dans la figure 6.1. Elles ont été trouvées par un petit programme.

Clauses faciles

Si on prouve, en supposant I vraie, que a_j préserve toutes les variables et toutes les fonctions d'état qui interviennent directement dans I_i, on en déduit facilement que P_{ij} est vraie. Or il est relativement facile de prouver sous l'hypothèse I les propriétés suivantes :

- A, Af, Obj et Cf sont préservées par toutes les actions sauf 1, 21 et 43
- Wh n'est modifiée que par 2, 4, 14, 23, 24, 30, 40, 43 et 50.
- Gr par 14, 23, 24, 30, 40, 50 et 51.
- Bk par 14, 20, 21, 23, 24, 30, 40, 50 et 51.
- Bu par 1, 2, 4 et 20.
- $\mathsf{FreeList}_m$ par 18, 19 et 20.
- Fr par 1, 3, 4, 20 et 63.
- $\mathsf{Creating}_m$ par 20, 21 et 26.
- Val et C par 3, 4, 26 et 63.
- Hp et P par 3, 4, 26, 33 et 63.
- Acc_m par 8, 13, 14, 16, 17, 26, 27 et 33.
- Kill_m par 27, 28, 29, 31, 32 et 33.
- Done par 14, 21, 23, 24, 30, 40 et 52.
- FDone par 14, 21, 23, 24, 30, 40 et 53.
- Tr par 3, 4, 14, 26, 28–33, 40, 52 et 63.
- Reach par 11, 13, 24–33, 40, 49, 51–54 et 63.
- Mrk par 11, 14, 24, 26–33, 40, 49, 51, 53 et 63.
- Q et L_m par 6 et 8.
- N_m par 20–24.
- $\mathsf{Val} \setminus \mathsf{Wh}$ par 14, 26, 30, 40 et 50.

Par exemple, on montre que a_{14} préserve Bu par le raisonnement suivant : si x appartient à *toMark* alors il appartient à Acc_m (d'après la définition de Acc_m), donc à Val (d'après I_4), donc il n'appartient pas à Bu (d'après les définitions de Val et Fr). L'action a_{14} change la couleur de x en gris. Celui-ci n'était pas bleu avant l'action et il n'est pas bleu après, et l'action ne change pas la couleur d'autres objets, donc Bu ne change pas.

Les 406 P_{ij} que l'on prouve grâce aux propriétés précédentes sont notées par $\times$ dans la table de la figure 6.1. Les propriétés ont été prouvées à la main, et les P_{ij} ont été déduites par un deuxième petit programme.

Clauses "difficiles"

Il nous reste 852 propositions à prouver (notées par $\bullet$ dans la table). Nous ne les détaillerons pas toutes ici. Certaines de ces propositions sont encore très faciles à prouver car l'invariant est une implication dont l'hypothèse est fausse quand la précondition de l'action est vraie, et reste fausse quand l'action s'exécute. Par exemple $P_{13,j}$ pour $j \in \{39, 50, 51, 57, 59, 62, 63\}$.

D'autres sont faciles à prouver par un argument de monotonie. Par exemple, $P_{18,54}$: l'action 54 retire un élément de *toTrace* et préserve Val. Si *toTrace* est inclus dans Val avant l'action, il le sera encore après.

Certaines des P_{ij} sont plus difficiles à prouver. Par exemple $P_{4,63}$: l'action 63 retire de Val tous les objets blancs compris entre 0 et *ptr*. Il faut prouver qu'aucun objet blanc ne se trouve dans $X^*(\bigcup \mathsf{Acc}_m)$ quand cette action se déclenche.

I_6 nous donne $status_C = phase = Async$ car $step = Scan$. On en déduit grâce à I_{26} que pour tout m, $status_m \in \{Async, Dead, Avail, Quick\}$. Si $status_m = Async$ alors I_{31} nous donne $\mathsf{Acc}_m \subset \mathsf{Mrk}$. Sinon, I_{32}, I_{33} et I_{34} nous donnent $\mathsf{Acc}_m = \emptyset$.

On a donc $\bigcup \mathsf{Acc}_{Pid} \subset \mathsf{Mrk}$. I_{21} nous dit que Mrk est stable par P, donc $\mathsf{P}^*(\bigcup \mathsf{Acc}_{Pid}) \subset \mathsf{Mrk}$. Il suffit donc de prouver que $\mathsf{Mrk} \subset \mathsf{Val}\backslash\mathsf{Wh}$, on aura alors $\mathsf{Mrk} \subset \mathsf{Val}'$. À cause de la définition de Mrk, il suffit donc de prouver que $\mathsf{Tr}^*(\mathsf{Reach}) = \emptyset$.

On prouve pour cela que $\mathsf{Reach} = \emptyset$: la précondition de l'action 63 est vérifiée, donc *toBlack*, *toTrace* et *cache* sont vides ; pour tout m, $toMark_m$ est vide (I_{30} si $status_m = Async$; I_{32}, I_{33} et I_{34} sinon) et $marking_m$ est faux (I_6 donne $status_C = phase = Async$ car $Step = Scan$, I_{26} et I_{32} donnent $pc = Halt \vee status_m = status_C$, et I_{29} permet de conclure) ; $\mathsf{Gr}\backslash\mathsf{ScanDone}$ est vide à cause de I_{24} (Gr est inclus dans $\mathsf{Mrk}\backslash\mathsf{Done}$ et $ptr = limit$) ; enfin, $\mathsf{ReachFields} = \emptyset$ car $fields = \emptyset$.

On a donc prouvé que les objets accessibles par les mutateurs sont tous marqués lorsque le GC passe de l'étape *Scan* à l'étape *Sweep*. Il reste à prouver que les objets en cours d'allocation le sont aussi, c'est-à-dire $\bigoplus \mathsf{Creating}_{Pid} \subset \mathsf{Obj}\backslash\mathsf{Fr}$. Cela se fait facilement avec I_{39} pour les mutateurs vivants ($status_m = Async$) et avec I_{32} et la définition de $\mathsf{Creating}_m$ pour les mutateurs morts ($status_m \in \{Dead, Avail, Quick\}$).

Chapitre 7

GC concurrent avec une génération

Nous avons implémenté l'algorithme exposé dans les chapitres 4 et 5, avec un mécanisme de générations, pour une version parallèle de Caml Light, et une version incrémentale de cet algorithme pour Caml Light séquentiel.

La section 7.1 détaille la relation entre l'algorithme écrit en TLA et son implémentation en C. La section 7.2 explique le système de générations ajouté à cet algorithme. Dans la section 7.3, nous donnons les résultats expérimentaux obtenus avec ce système concurrent à générations. La section 7.4 décrit la version incrémentale de cet algorithme, qui est utilisée dans Caml Light versions 0.5 à 0.7.

7.1 Implémentation de l'algorithme concurrent

7.1.1 Concurrent Caml Light

Concurrent Caml Light (CCL) est une version du système Caml Light modifiée pour fonctionner sur machines parallèles à mémoire partagée. L'organisation des programmes CCL reflète celle de la machine : un programme CCL est composé de plusieurs tâches séquentielles qui fonctionnent dans une mémoire partagée.

Le langage de programmation de CCL est le même que celui de Caml Light. Les primitives de parallélisme sont fournies au programmeur au moyen du module `thread`, implémenté par le système, qui donne des fonctions pour gérer les processus, les verrous d'exclusion mutuelle et les conditions d'attente. Nous allons décrire brièvement ces primitives, qui sont un sous-ensemble de celles fournies par la bibliothèque *C Threads* du système Mach, ou par la bibliothèque *POSIX Threads* du standard POSIX.

La figure 7.1 donne l'interface Caml Light des primitives de parallélisme. Cette interface déclare trois types abstraits : `process`, `mutex` et `condition`. Le type `process` permet à un programme de manipuler ses propres processus. Les fonctions qui agissent sur les processus sont les suivantes :

`fork` prend en argument une fonction f et lance un nouveau processus pour exécuter

```
type process;;
value fork : (unit -> 'a) -> process
  and exit : unit -> 'a
  and join : process -> unit
  and detach : process -> unit
  and yield : unit -> unit
  and self : unit -> process
;;
type mutex;;
type condition;;
value new_mutex : unit -> mutex
  and new_condition : unit -> condition
  and lock : mutex -> unit
  and unlock : mutex -> unit
  and try_lock : mutex -> bool
  and signal : condition -> unit
  and broadcast : condition -> unit
  and wait : condition -> mutex -> unit
;;
```

Figure 7.1: L'interface du module `thread`

la fonction f. Le résultat retourné par f sera ignoré : la communication entre processus ne peut se faire que par effets de bord. La fonction `fork` retourne un objet de type `process` qui est le descripteur du nouveau processus.

`exit` permet à un processus de s'arrêter prématurément. Un processus p s'arrête normalement quand la fonction f (passée en argument à `fork` pour créer p) termine son calcul.

`join` permet à un processus p_1 d'attendre l'arrêt d'un processus p_2. L'argument de `join` est le descripteur de p_2. La fonction `join` bloque p_1 jusqu'à l'arrêt de p_2. Si p_2 s'est déjà arrêté au moment où p_1 appelle `join`, celle-ci retourne immédiatement. La fonction `join` rend invalide le descripteur de p_2, donc il ne faut l'appeler qu'une fois pour un processus p_2 donné.

`detach` signale au système qu'on n'appellera jamais `join` sur le processus p passé en argument. Cette fonction sert à faciliter le travail de la bibliothèque de tâches.

`yield` ne fait rien dans une implémentation préemptive des tâches (ce qui est pratiquement toujours le cas sur machines parallèles).

`self` permet à une tâche d'obtenir son propre descripteur.

Le type `mutex` est celui des verrous d'exclusion mutuelle. Les fonctions qui agissent sur les verrous sont les suivantes :

`new_mutex` permet d'allouer un nouveau verrou.

`lock` permet à un processus de prendre un verrou. Un seul processus à la fois peut prendre un verrou donné. Si le verrou passé en argument à `lock` est déjà pris par un autre processus, `lock` attend que celui-ci rende le verrou.

`unlock` permet au processus de rendre un verrou qu'il a pris.

`try_lock` fonctionne comme `lock` mais n'attend pas : si le verrou est déjà pris, cette fonction retourne `false`. S'il n'est pas pris, `try_lock` prend le verrou et retourne `true`.

Lorsqu'un processus doit attendre un événement particulier, il peut entrer dans une boucle qui teste sans arrêt si l'événement s'est produit, mais cette solution (que l'on appelle *attente active*) est très inefficace : elle oblige un processeur à exécuter cette boucle qui consomme du temps de calcul.

Le type `condition` permet d'éviter l'attente active : un objet de type `condition` est en quelque sorte une salle d'attente pour les processus. Le programmeur définit une condition associée à chaque événement. Un processus qui doit attendre un événement va se mettre en attente sur la condition (il entre dans la salle d'attente). Un processus qui déclenche l'événement "signale" la condition (il appelle le suivant dans la salle d'attente).

L'attente sur la condition se fait par un appel système spécial qui permet au processus de se bloquer sans consommer de temps de calcul. L'utilisation des verrous et des conditions est décrite plus en détail dans [17, 25].
Il y a quatre fonctions qui manipulent les conditions :

`new_condition` alloue une nouvelle condition.

`signal` prend en argument une condition c et réveille au moins un des processus qui sont en attente sur c (s'il y en a).

`broadcast` fonctionne comme `signal` mais réveille tous les processus en attente sur c.

`wait` prend en argument une condition c et un verrou m, qui doit être déjà pris par le processus qui appelle `wait`. Cette fonction va atomiquement rendre le verrou et se placer en attente sur la condition.

Ces déclarations sont simplement une traduction en ML d'un sous-ensemble de la bibliothèque *C Threads*. Ce sont des primitives de bas niveau que nous utiliserons pour implémenter des mécanismes plus faciles à utiliser, tels que des canaux de communication, des futurs [47], des événements à la CML [82], etc. Ces mécanismes de haut niveau seront implémentés en Caml, ce qui nous évite de compliquer outre mesure l'interpréteur de byte-code et le GC : en implémentant les autres modèles de programmation parallèle au-dessus du module `thread`, nous n'avons à résoudre qu'une seule fois les problèmes posés par l'ajout de parallélisme à l'interpréteur de byte-code.

Ces problèmes se réduisent à l'adaptation du GM, car chaque primitive ML appellera simplement la primitive C correspondante. Nous utilisons le GC concurrent décrit dans les chapitres 4 et 5 et nous lui ajoutons une génération locale à chaque processus.

7.1.2 De TLA à C

L'algorithme donné dans le chapitre 5 n'est pas directement utilisable car il n'existe pas de compilateur pour TLA. Il faut donc traduire les structures de données de l'algorithme (notamment les ensembles et les multi-ensembles) en structures de données utilisables par C (listes chaînées, etc.), et il faut traduire les actions en instructions C.

Traduction des structures de données

Nous distinguerons quatre sortes de types, selon les problèmes qu'ils posent pour la traduction en C :

- Les types énumérés (*Bool*, *Colors* et *Statuses*) : ils ne posent pas de problème particulier, on peut les représenter par des entiers car l'algorithme n'a pas besoin de les distinguer des entiers (ils n'apparaissent jamais dans une union disjointe avec **Nat**). Les types *Steps* et *Labels* sont, dans le modèle, des types énumérés, mais ils ne servent qu'à spécifier des contraintes sur l'ordre d'exécution des actions. Dans le programme C, ces contraintes vont être codées par le séquencement des instructions de chaque processus.

- La mémoire est représentée dans le modèle par un tableau, c'est-à-dire une fonction de *Addr* dans *Words*. C'est bien entendu la mémoire de la machine qui joue ce rôle dans le programme C. Le type *Addr* représente donc le type des pointeurs de C.

- Les types *Addr* et *Sizes* sont définis dans le modèle comme égaux à l'ensemble **Nat** des entiers naturels. Malheureusement, le langage C n'a pas de type correspondant à **Nat** : les variables entières prennent leur valeur dans un intervalle (en général $[0, 2^{32}[$) et les calculs sont effectués modulo la taille de cet intervalle. Pour être sûr que le programme fonctionne correctement, il faut donc prouver que les calculs effectués ne déborderont pas de cet intervalle.

  Pour ce faire, nous prouvons que tous les entiers manipulés par le modèle sont majorés par *top*, sauf bien sûr dans l'action 1, qui incrémente *top*. Dans notre programme, cette action devra vérifier que *top* ne déborde pas. Si *top* déborde, alors l'espace d'adressage de la machine est épuisé, et le programme doit s'arrêter en signalant l'erreur "pas assez de mémoire". Tant que cette erreur ne se déclenche pas, les calculs effectués par notre programme C coïncident avec ceux du modèle formel.

- Les ensembles ou multi-ensembles correspondent généralement à des structures de contrôle du programme : boucles (*fields* représente l'ensemble des valeurs restant à parcourir par l'indice de boucle) ou variables locales d'une procédure récursive (par exemple *toTrace*), etc.

Il y a deux exceptions à ce dernier cas : la liste libre et les racines des mutateurs. Le multi-ensemble des racines des mutateurs représente toutes les structures de données du mutateur qui peuvent contenir des pointeurs vers le tas. C'est un multi-ensemble car nous voulons éviter d'imposer des contraintes d'unicité aux mutateurs. La seule contrainte est que le mutateur doit pouvoir énumérer ses racines (pour le marquage), ce qui correspond à l'affectation $toMark \leftarrow toMark \oplus roots$ de l'action 11. Si l'énumération des racines est effectuée par une boucle, *toMark* représentera l'ensemble des valeurs restant à parcourir par l'indice de boucle, et le corps de la boucle sera constitué des actions 13 et 14.

Dans notre modèle formel, la liste libre est un multi-ensemble, mais l'invariant I_3 affirme que ce multi-ensemble ne contient pas d'éléments dupliqués (car il est inclus dans un ensemble). Nous pouvons donc utiliser une implémentation classique par liste chaînée pour la liste libre : chaque élément contient un pointeur vers le suivant. Nous gardons les éléments de cette liste par ordre des adresses croissantes, pour les raisons suivantes :

- Pour faciliter l'implémentation de l'action 36 : le collecteur utilise un pointeur vers le premier objet de la liste libre situé après *sublimit*. Il peut donc facilement enlever de la liste libre les objets situés après *sublimit*.
- Le GC repère les objets libres par son balayage, donc il les trouve dans l'ordre des adresses croissantes. Il lui est facile de les insérer au bon endroit dans la liste libre.

Traduction des actions

L'algorithme écrit en TLA comporte un non-déterminisme important : dans la plupart des états, il y a plusieurs actions activées. Ce non-déterminisme se traduit à deux niveaux : l'entrelacement arbitraire des actions des mutateurs et du collecteur nous permet de faire exécuter chaque processus sur un processeur différent, sans synchronisation (sauf pour les quelques actions qui nécessitent explicitement une synchronisation). C'était le but premier de notre système.

Chaque processus comporte aussi un certain degré de parallélisme. Par exemple, le collecteur peut exécuter l'action 46 à tout moment, ou les actions 47, 48 et 49 tout au long de ses étapes *Mark* et *Scan*. Ce parallélisme explicite dans le modèle nous permet de savoir quelles sont les actions qui doivent se dérouler dans un ordre précis, et il nous permet de choisir l'ordre dans lequel nous écrivons les autres. Il suffit que le code C corresponde à un ordre d'exécution possible du modèle pour qu'il soit correct (à condition bien sûr qu'il respecte les contraintes d'équité).

Nous pourrions donc par exemple utiliser ce degré de liberté pour paralléliser le collecteur, c'est-à-dire le diviser en plusieurs processus. En examinant le modèle, on

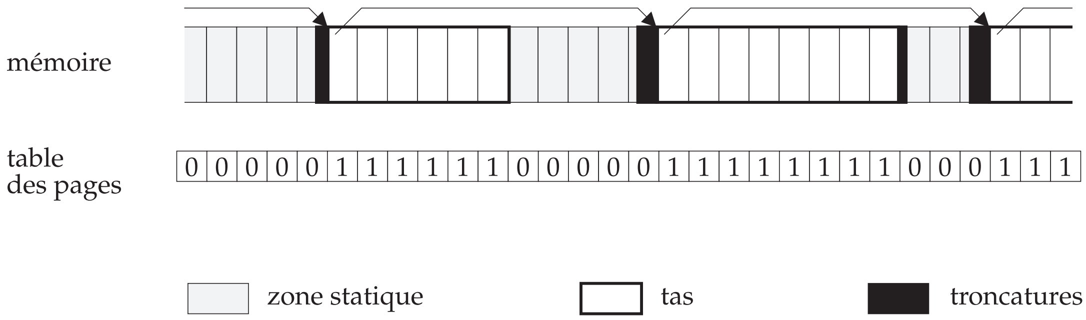


Figure 7.2: Chaînage des morceaux du tas

constate qu'il est plus facile de paralléliser le marquage que le balayage (contrairement à ce qui est dit dans [37]). On peut aussi réarranger le code du collecteur (notamment le marquage) pour changer l'ordre des accès à la mémoire, dans le but d'améliorer la localité des références. Cette modification s'avérera sans doute payante sur les machines à mémoire virtuelle partagée, dans lesquelles un accès à la mémoire coûte beaucoup moins cher s'il se fait à une adresse proche de l'accès précédent.

7.1.3 Quelques problèmes d'implémentation

Organisation de la mémoire

Pour améliorer la compatibilité avec `malloc`, nous avons changé l'organisation de la mémoire par rapport à Caml Light version 0.4 (cf. section 3.3.2).

Au lieu d'une zone contiguë de mémoire, le tas est constitué de *morceaux*. Un morceau est un bloc de mémoire contigu, muni d'un en-tête qui donne sa taille. La mémoire située entre les morceaux constitue la zone statique ; elle est directement gérée par `malloc`. Le GM alloue les morceaux en appelant `malloc`, ce qui assure une parfaite compatibilité : le GM de Concurrent Caml Light est vu par `malloc` comme un simple client.

Pour que le balayage reste possible, il faut placer les morceaux dans une liste chaînée. De plus, cette liste doit être triée par ordre d'adresses croissantes car *swept* ne doit jamais décroître lors du balayage, sinon les actions de *CreateProc* pourraient donner la mauvaise couleur aux objets créés (la même remarque s'applique à *scanned* et *UpdateProc*).

Pour que le marquage reste possible, il faut pouvoir distinguer un pointeur vers le tas (que le GC doit suivre) d'un pointeur hors du tas (que le GC doit ignorer). Pour que le marquage reste raisonnablement efficace, il faut que ce test soit rapide. Pour cela, nous découpons la mémoire en pages de taille fixe (8 kilo-octets) alignées (l'adresse du début d'une page est toujours un multiple de 8 kilo-octets). Nous faisons en sorte que chaque morceau soit constitué d'un nombre entier de pages en le tronquant aux deux bouts si nécessaire avant de l'insérer dans la liste chaînée. Les pages sont numérotées : l'adresse de la page, divisée par la taille d'une page, donne le numéro de la page. En pratique, on utilise toujours une puissance de deux pour la taille de la page, et on trouve donc le numéro de page par un simple décalage arithmétique.

Le GM maintient un tableau de booléens (la *table des pages*) qui donne pour chaque page son appartenance au tas. Ainsi, étant donné un pointeur on peut déterminer si c'est un pointeur vers le tas en effectuant un décalage (pour obtenir le numéro de la page) et un accès à un tableau (la table des pages). Cette technique n'est pas beaucoup plus coûteuse que les deux comparaisons de l'ancien GC (pour savoir si l'adresse se trouve dans un intervalle), et elle nous évite beaucoup de problèmes avec `malloc`. La figure 7.2 illustre les structures de données que nous venons de décrire.

Les agrandissements du tas sont gérés par une fonction qui implémente l'action 1 du modèle. Cette fonction appelle `malloc` pour obtenir un nouveau bloc de mémoire, crée un objet bleu à l'intérieur de ce bloc, ajoute le bloc dans la liste chaînée des morceaux du tas et l'objet bleu dans la liste libre et marque dans la table de pages les pages correspondant à ce nouveau morceau du tas. Cette fonction utilise un verrou qui protège l'accès à la liste libre et au chaînage des morceaux du tas.

Implémentation du module `thread`

Chaque fonction du module `thread` utilise la fonction correspondante de la bibliothèque *C Threads*. Les fonctions `detach`, `yield`, `self`, `try_lock`, `unlock`, `signal` et `broadcast` ne posent aucun problème : la fonction ML se contente d'appeler la fonction C correspondante.

La bibliothèque C donne quatre fonctions pour allouer et désallouer les verrous : `mutex_alloc`, `mutex_init`, `mutex_clear` et `mutex_free`. Les fonctions `mutex_alloc` et `mutex_free` permettent d'allouer et de désallouer (en zone statique) la structure de données qui contiendra un verrou ; `mutex_init` permet d'initialiser un verrou dans cette structure de données, et `mutex_clear` permet de supprimer le verrou. Cette dernière fonction sert à libérer les ressources du système réservées par `mutex_init`.

Notre fonction `new_mutex` utilise ces quatre fonctions de la façon suivante : elle appelle `mutex_alloc` et `mutex_init` pour obtenir un nouveau verrou en état de marche, puis elle alloue un objet dans le tas (que nous appellerons un *talon*), qui contiendra un pointeur vers ce verrou. C'est ce talon qui est retourné par `new_mutex`. Le but de cette indirection supplémentaire est de permettre au GC de détecter que le verrou est devenu inaccessible. En effet, le verrou lui-même se trouve dans la zone statique, hors de la juridiction du GC. Le talon se trouve dans le tas, donc le GC sait quand il devient inaccessible, et c'est le seul objet qui contienne un pointeur vers le verrou, donc si le talon est inaccessible alors le verrou l'est aussi. Lorsque le GC désalloue le talon, il devra aussi désallouer le verrou. Pour obtenir ce résultat, nous avons implémenté un mécanisme d'objets *finalisés* : ce sont des objets qui portent un *type d'objet* spécial (`final_tag`), et dont le premier champ est un pointeur vers une fonction C (la fonction de *finalisation*). Lorsque l'étape de balayage trouve un objet blanc portant le type `final_tag`, elle appellera la fonction de finalisation de cet objet avant de l'ajouter à la liste libre. Dans le cas des verrous, cette fonction de finalisation appelle `mutex_clear` et `mutex_free`. Nous utilisons pour allouer et désallouer les conditions le même mécanisme que pour les verrous.

Il reste les fonction `fork` et `exit`, qui doivent implémenter les actions 5 à 9. La fonction `fork` crée un nouveau processus (en appelant la fonction `fork` de *C Threads*) et lui donne à exécuter le code de son argument. La fonction `exit` doit implémenter l'action 9 : il faut vider la liste libre locale et signaler au GC la mort du processus avant de stopper le processus (avec la fonction `exit` de *C Threads*).

La procédure *Cooperate*

La version séquentielle de l'interpréteur de byte-code comportait déjà une fonction qui était appelée régulièrement. Nous utilisons cette fonction pour garantir que *Cooperate* est appelée assez souvent. Cependant, l'interpréteur de byte-code peut se bloquer lors de certains appels système (`read`, `write`, etc.) ou lorsqu'il exécute des primitives de synchronisation (les fonctions `join`, `lock` et `wait` du module `thread`).

C'est pourquoi nous associons à chaque processus un drapeau protégé par un verrou qui signale si le processus est bloqué dans une de ces fonctions. Lors des poignées de mains, le collecteur appelle lui-même la procédure *Cooperate* pour les processus qui sont bloqués. Le verrou nous permet de garantir que le processus ne va pas se réveiller pendant cet appel à *Cooperate*.

7.2 Ajout d'une génération

Les résultats présentés au chapitre 3 montrent l'intérêt d'ajouter une génération avec un GC mineur à copie. Cependant, il est difficile de faire fonctionner un GC à copie concurrent car le GC doit déplacer les objets pendant qu'ils sont utilisés par les autres processus. Nous résolvons ce problème en utilisant un tas mineur par processus, qui ne contient que des objets inaccessibles pour les autres processus. Le GC mineur peut alors interrompre un processus pour collecter son tas mineur, sans interférer avec le travail des autres processus.

7.2.1 Génération mineure locale

L'organisation de la mémoire est illustrée par la figure 7.3. Chaque processus possède une pile (qui contient ses racines) et un tas mineur, qui se trouvent dans sa mémoire locale. Le processus alloue dans son tas mineur ; quand celui-ci est plein, le mutateur s'arrête et le GC mineur se déclenche. Le GC mineur utilise l'algorithme à copie classique pour recopier tous les objets vivants du tas mineur dans le tas majeur et mettre à jour les racines.

Le GC mineur est asynchrone : il peut se déclencher à tout moment, sans se synchroniser avec les autres processus. Comme il déplace les objets du tas mineur, il faut que ces objets soient inaccessibles aux autres mutateurs (sinon ils pourraient utiliser l'ancienne copie, pourtant invalide). En conséquence, nous devons faire en sorte qu'aucun pointeur venant de l'extérieur (le tas majeur ou les autres processus) ne pointe vers la mémoire

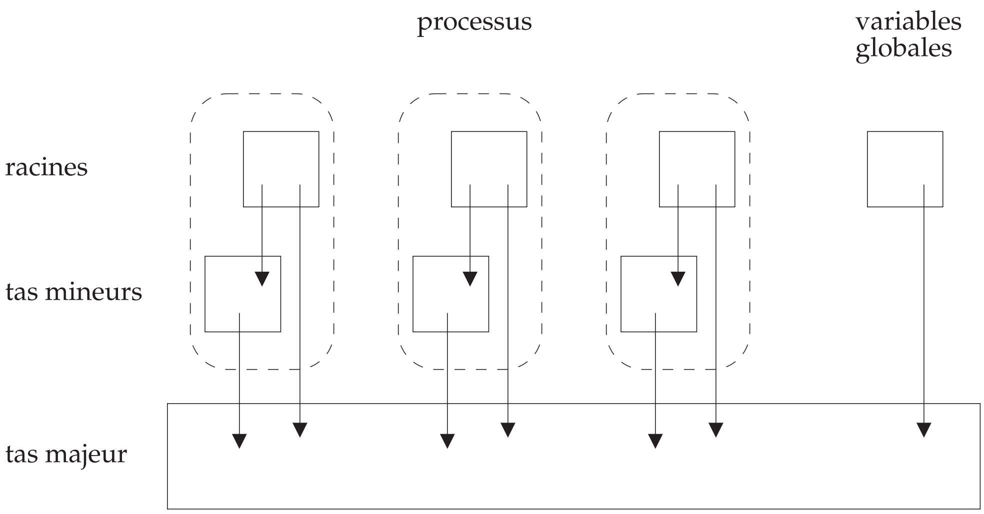


Figure 7.3: Organisation de la mémoire

locale d'un processus. Les mécanismes que nous utilisons pour maintenir cette propriété sont détaillés dans les deux sections suivantes.

7.2.2 Copie à la modification

Les pointeurs vers la mémoire locale d'un processus (en gris sur la figure 7.4) sont interdits dans notre système. De tels pointeurs ne peuvent être créés que par une affectation (dans une variable globale ou dans le tas) ou par le passage d'arguments à un nouveau processus (fonction `fork`).

Pour éviter cette création de pointeurs interdits, notre procédure de modification (qui doit déjà gérer la coopération avec le GC majeur) va, lorsque l'objet affecté se trouve dans le tas majeur et la nouvelle valeur dans le tas mineur, recopier la nouvelle valeur (et tous ses descendants mineurs) dans le tas majeur et utiliser un pointeur vers cette copie pour son affectation. Cette recopie est similaire à celle effectuée par le GC mineur, mais la procédure de modification ne peut pas mettre à jour tous les pointeurs vers l'objet copié (car elle ne parcourt pas toutes les racines, ni le tas mineur). Il faut donc laisser en place la valeur d'origine : la recopie duplique l'objet au lieu de le déplacer.

Cette duplication est sémantiquement correcte : un programme ML n'a aucun moyen de distinguer l'original de la copie (sauf pour les objets mutables, que nous traitons dans la section suivante). Il faut cependant mettre en place un pointeur de renvoi, pour deux raisons :

- Si une autre modification veut copier le même objet, elle pourra ainsi utiliser la copie déjà faite.
- Le prochain GC mineur n'aura pas à copier cet objet (s'il est encore vivant) : il pourra lui aussi utiliser directement la copie.

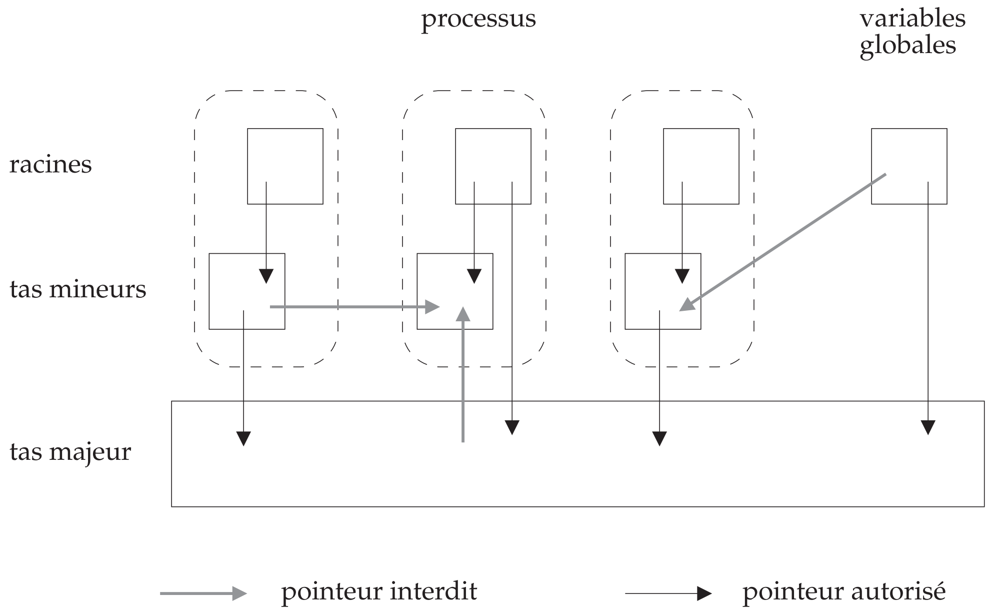


Figure 7.4: Isolement des mémoires locales

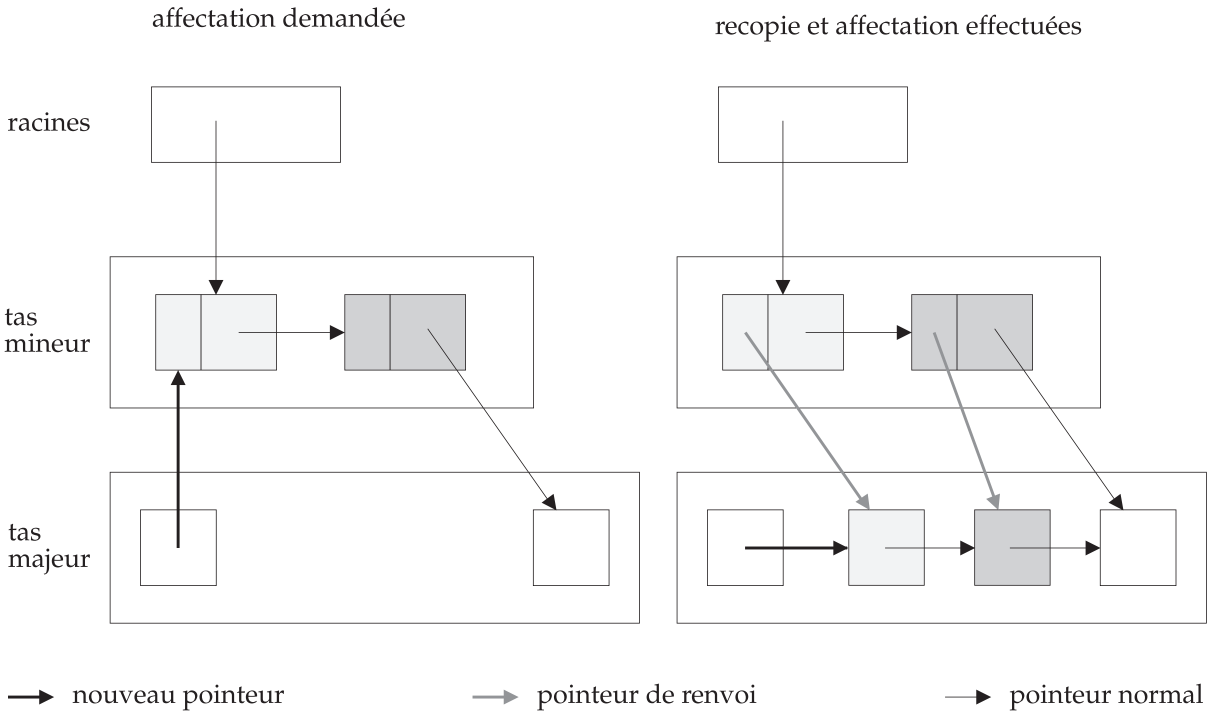


Figure 7.5: Modification dans le tas majeur

Dans un GC à copie traditionnel, on stocke le pointeur de renvoi à la place du contenu de l'objet. Nous ne pouvons pas utiliser cette technique puisqu'il faut préserver le contenu de l'objet copié, qui est toujours vivant. Nous réservons donc, au moment d'allouer un objet dans le tas mineur, un mot de mémoire (placé juste avant l'objet) pour le pointeur de renvoi.

Le travail de copie effectué par la procédure de modification est illustré par la figure 7.5. La modification des variables globales et le passage d'arguments à un nouveau processus utilisent la même technique.

7.2.3 Allocation des objets mutables

Nous avons vu que la primitive de modification doit dupliquer les objets mineurs. Cette duplication n'est pas gênante pour les objets non mutables (le programme ne peut pas distinguer les deux copies car le langage ML ne permet pas de tester l'égalité de deux pointeurs), mais pour les objets mutables elle ne fonctionne pas : si un objet mutable est dupliqué et si un processus effectue une affectation sur l'une des copies, l'autre copie n'est pas mise à jour et les autres processus risquent de lire la mauvaise valeur.

Notre solution consiste à obliger les mutateurs à allouer leurs objets mutables directement dans le tas majeur, puisque les objets majeurs ne sont jamais dupliqués. Cela implique une modification mineure du compilateur : grâce au typage statique de ML, le compilateur sait toujours s'il alloue un objet mutable ou non.

Cette allocation directe dans le tas majeur est coûteuse, puisqu'elle court-circuite le mécanisme de générations. Heureusement, un programme ML typique alloue peu d'objets mutables, et de plus les objets mutables ont souvent une durée de vie plus grande que les objets non mutables.

Notre système de générations repose donc de façon essentielle sur deux caractéristiques de ML :

- La sémantique du langage est préservée par la duplication des objets non mutables.
- Le compilateur distingue les objets mutables des objets non mutables, ce qui nous permet de les traiter différemment.

On voit donc que le typage statique n'est pas seulement un outil puissant au service du programmeur : c'est aussi une analyse statique du programme dont le résultat sert à implémenter une technique de GC performante.

7.2.4 Interface avec le GC majeur

En ce qui concerne le GC majeur, l'ensemble mutateur + GC mineur constitue un mutateur et le tas mineur fait partie des racines. Lorsque le GC majeur demande les racines (par la troisième poignée de mains), le mutateur les lui donne en effectuant un GC mineur qui grise les objets pointés par les racines et qui recopie et grise les objets vivants du tas mineur.

Programme	Knuth-Bendix	Pipeline	Parallèle	SIMPLE
Nombre de processus	15	3	12	6,4 (en moyenne)
Charge du GC majeur	32 %	16 %	39 %	10 %
Copie à la modification	96 %	43 %	36 %	96 %
GC mineur (moyenne)	2,9 ms	2,3 ms	6,3 ms	2,1 ms
GC mineur (maximum)	64 ms	180 ms	110 ms	360 ms
Modification (moyenne)	260 μs	37 μs	55 μs	70 μs
Modification (maximum)	70 ms	6,9 ms	31 ms	20 ms
Liste libre (moyenne)	60 μs	19 μs	1,6 ms	54 μs
Liste libre (maximum)	17 ms	220 μs	110 ms	25 ms

Figure 7.6: Performances du GC concurrent

Dans notre modèle formel, cela correspond au cas où *marking* est vrai et *answering* est faux, donc à partir de l'exécution de l'action 11. Le GC mineur peut créer des objets : la recopie d'un objet mineur dans le tas majeur consiste à créer un nouvel objet dans le tas majeur et à lui donner le contenu de l'objet mineur. Le GC mineur peut aussi supprimer l'ancien objet du tas mineur : les pointeurs qu'il contient (qui constituent des racines selon le GC majeur) disparaissent de *roots* (action 14) et de *toMark* (action 25), et la copie doit être marquée (action 26).

7.3 Performances du GC concurrent à générations

Cette section reprend les résultats de [37].

Nous avons utilisé une machine parallèle de type Encore Multimax, avec 14 processeurs (NS32532). Chaque processeur représente une puissance de calcul d'environ 6 MIPS, et est muni d'un cache de 256 kilo-octets. Nous avons utilisé une taille de tas mineur de 32 kilo-octets, qui rentre aisément dans le cache avec le programme et l'interpréteur de byte-code.

Nous avons utilisé les programmes suivants :

Knuth-Bendix Une version parallèle du programme KB utilisé dans la section 3.4.1. Le programme utilise quinze processus, qui communiquent par des files de messages implémentées par des structures de données mutables. La quantité de communication entre processus est très grande.

pipeline Une version en pipeline du compilateur Caml Light. Un processus pour l'analyse lexicale, un pour l'analyse syntaxique, et un pour le reste de la compilation. La quantité de communication est plus faible que pour Knuth-Bendix, mais elle reste assez importante.

parallèle Une version parallèle du compilateur Caml Light. Cette version compile plusieurs fichiers en même temps, chaque fichier étant compilé séquentiellement par un processus séparé. Il y a très peu de communication entre les processus.

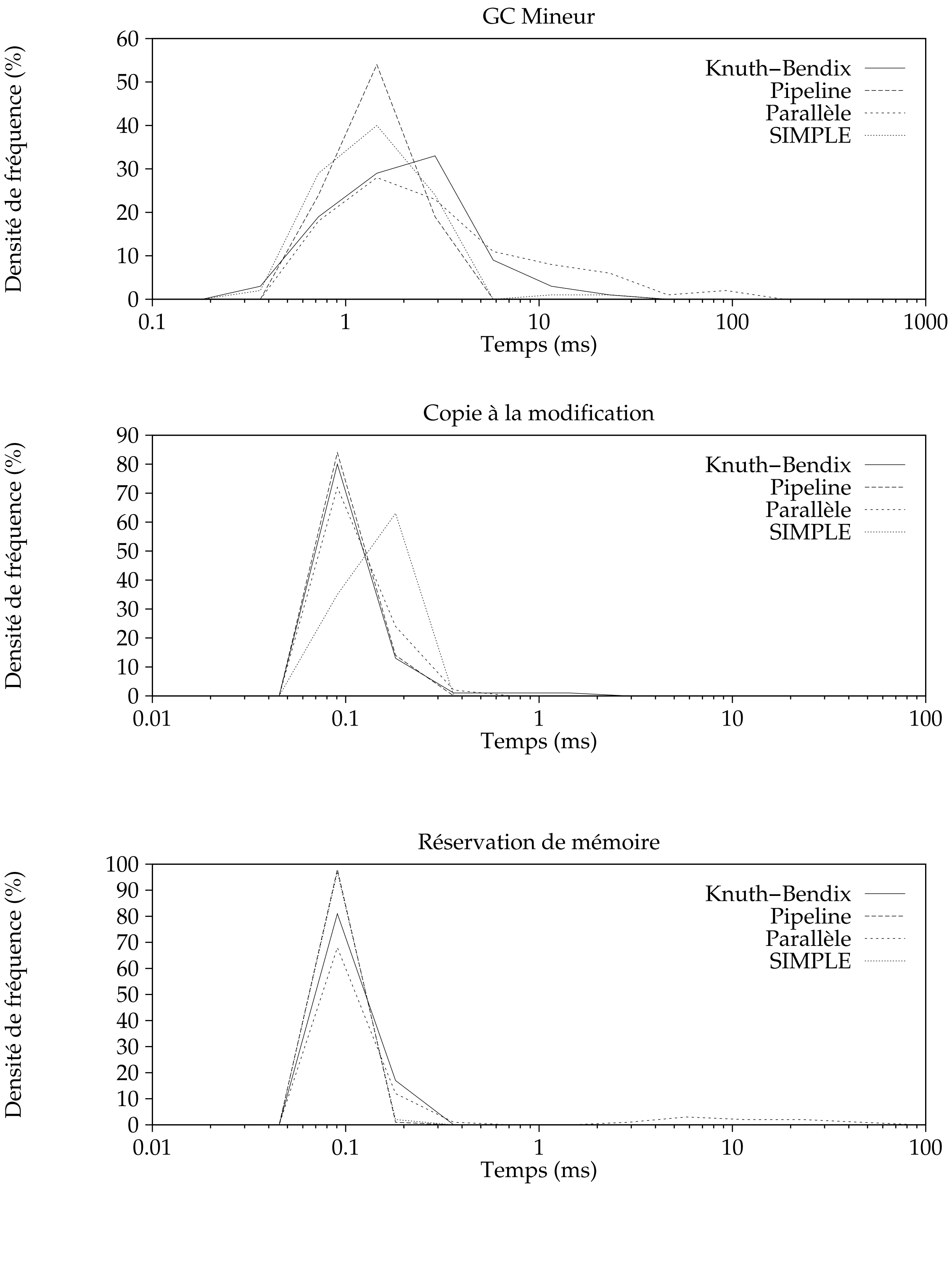


Figure 7.7: Distribution des temps de latence

SIMPLE Le programme SIMPLE (cf. section 3.4.1), parallélisé par Morrisett et Tolmach [75]. Ce programme utilise énormément de tableaux mutables.

Nous avons d'abord essayé de déterminer si le GC majeur travaille assez vite pour suivre le taux d'allocation total d'un grand nombre de mutateurs. Pour cela, nous utilisons le fait que le GC majeur ne fonctionne pas en continu : il se déclenche quand la taille de la liste libre tombe en dessous d'un certain seuil (dans nos expériences, ce seuil était fixé à 15% de la taille du tas). Nous mesurons donc la proportion du temps pendant laquelle le GC majeur est actif, que nous appelons la *charge* du GC majeur. La charge nous donne une estimation du temps de calcul nécessaire au GC pour désallouer les objets aussi vite qu'ils deviennent inaccessibles.

D'après les résultats de la figure 7.6, la charge du GC majeur est entre 2 et 5 % par mutateur, ce qui signifie que notre système peut fonctionner avec 20 à 50 mutateurs, selon leur taux d'allocation. Avec un programme qui ne fait rien d'autre qu'allouer des objets mutables, il suffit de quatre mutateurs pour saturer le GC majeur. Nous avons donc la confirmation expérimentale que les programmes ML réalistes n'allouent pas beaucoup d'objets mutables.

Nous avons ensuite mesuré le pourcentage des modifications qui nécessitent une recopie d'objets. Il est très élevé pour Knuth-Bendix et SIMPLE. Pour Knuth-Bendix, c'est parce que le programme n'utilise les mutables que pour les communications entre processus. Pour SIMPLE, c'est parce que le programme effectue principalement des affectations de flottants dans des vecteurs. Les vecteurs sont vieux (car ils sont mutables) et les flottants sont jeunes (car ils sont calculés juste avant l'affectation).

Enfin, nous avons mesuré les temps de latence pour les trois opérations importantes du GM : le GC mineur, la recopie à la modification, et la réservation de mémoire. Pour chacune de ces opérations, nous donnons les latences moyenne et maximale dans la table 7.6, et la distribution des temps de latence sur la figure 7.7.

Les temps moyens sont remarquablement faibles. La plupart des GC mineurs se terminent en moins de 10 millisecondes et la modification reste raisonnablement efficace, même pour Knuth-Bendix, qui transmet de grosses structures de données entre les processus par modification d'objets mutables partagés. Enfin, la réservation utilise un verrou pour protéger l'accès à la liste libre globale. Un processus peut donc se trouver obligé d'attendre que ce verrou soit libre. Cette attente reste courte, sauf pour le compilateur parallèle.

On peut comparer ces résultats à ceux de [20] : avec un processeur trois fois moins rapide, nous obtenons des latences maximales comparable, et des latences moyennes 20 à 40 fois plus courtes. (Notons que le contexte d'utilisation du GC de [20] est différent du nôtre, puisqu'il s'agit d'un GC conservatif.) Nos résultats sont parfaitement suffisants pour un système interactif car les pauses sont généralement très courtes, mais pas pour un système temps-réel car il n'y a pas de garantie sur la durée des opérations : comme on le voit sur la figure 7.7, certaines des opérations prennent beaucoup plus longtemps que la moyenne (360 ms pour le GC mineur représente le cas le pire où il faut copier tout le contenu du tas mineur).

7.4 Version incrémentale

Depuis la version 0.5, Caml Light utilise un GC incrémental à générations qui est une version simplifiée du GC concurrent décrit dans les sections précédentes. La section 7.4.1 décrit la transformation du GC concurrent en GC incrémental et la section 7.4.2 présente quelques mesures de performance.

7.4.1 Description

Le principe du GC incrémental est simple. On restreint le programme à un seul mutateur, ce qui nous donne deux processus : le mutateur et le GC majeur. Pour que notre système fonctionne sur machines séquentielles, il faut fondre ces deux processus en un seul. On obtient ce résultat en transformant le GC majeur en coroutine incrémentale : le code séquentiel du GC majeur est transformé en une procédure qui effectue une partie du travail du GC avant de retourner. Cette procédure doit être capable de reprendre le travail là où elle s'est interrompue lors de l'appel précédent. Nous appellerons le travail exécuté par un appel à cette procédure une *tranche* de GC majeur. Le mutateur (c'est-à-dire l'ensemble mutateur + GC mineur) doit appeler de temps en temps cette procédure, que nous appellerons simplement le GC majeur.

Simplifications de l'algorithme

Comme le GC est incrémental, nous contrôlons entièrement l'entrelacement entre les actions du mutateur et du collecteur. Cela signifie que nous n'avons plus besoin de primitives de synchronisation. Par exemple, les sections critiques protégeant l'accès à la liste libre implémentent l'exclusion mutuelle en évitant de donner la main à l'autre "processus". Le collecteur reste bloqué pendant que le code du mutateur s'exécute, donc celui-ci n'a qu'à éviter d'appeler une fonction du collecteur pour empêcher le collecteur d'accéder à la liste libre. Inversement, le mutateur reste bloqué pendant l'exécution d'une fonction du collecteur, donc celui-ci peut accéder à la liste libre sans précautions particulières, à condition que cet accès soit terminé avant que la fonction ne termine.

Comme il n'y a qu'un mutateur, nous pouvons supprimer les deux premières poignées de mains. Comme la synchronisation est maintenant gratuite, nous supprimons aussi la troisième : lorsque le GC majeur demande les racines, le GC mineur les lui donne immédiatement et le collecteur n'a pas besoin d'attendre. De même, les tests approximatifs de *scanned* (action 31) et de *swept* (action 22) sont remplacés par des tests exacts puisque le collecteur ne peut pas changer ses variables pendant l'exécution de ces tests. De plus, le mutateur peut placer directement dans le cache les objets qu'il grise, puisque l'exclusion mutuelle avec le collecteur ne coûte rien. Le mutateur et le collecteur communiquent donc par le cache (au lieu des objets gris) et on ne positionne plus *dirty* qu'en cas de débordement du cache. Nous supprimons aussi l'étape *Clear* car le mutateur peut consulter la variable *limit* du collecteur et allouer directement en blanc les objets qui ne seront pas balayés.

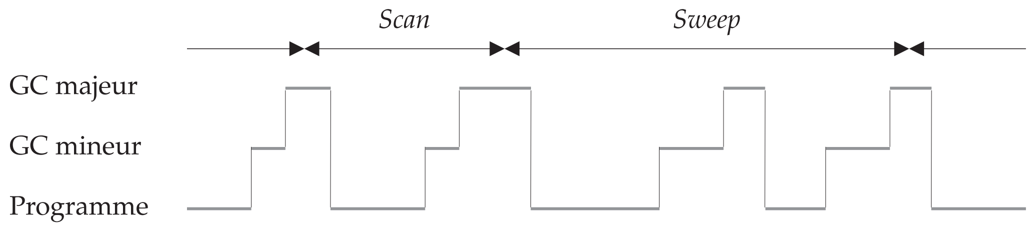


Figure 7.8: Entrelacement des "processus"

La restriction à un seul mutateur nous permet de revenir à un système de générations conventionnel : les pointeurs inverses sont de nouveau autorisés, nous ne faisons plus de copie à la modification et la fonction de modification doit mettre à jour une table de pointeurs inverses.

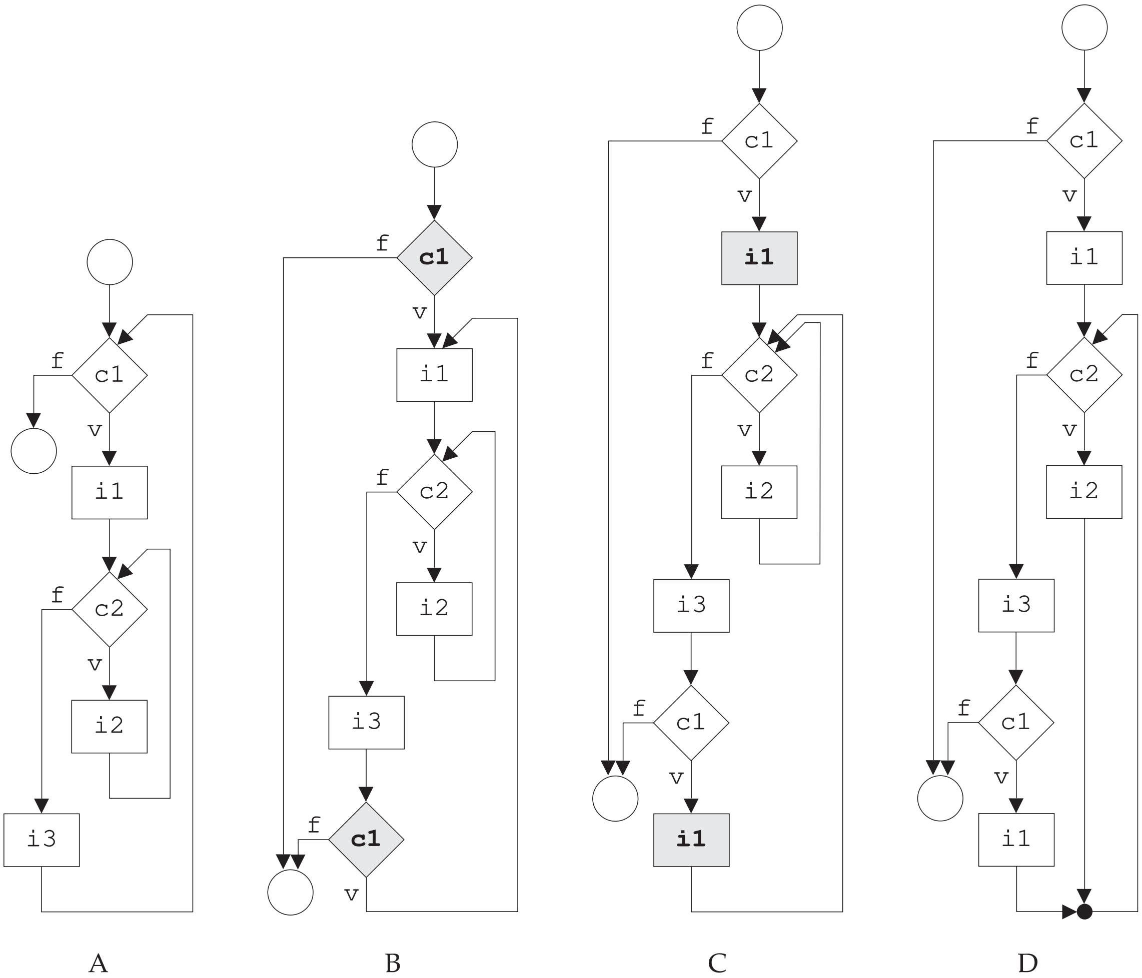
Le GC mineur est un événement périodique assez régulier et nous l'utilisons pour savoir quand appeler le GC majeur : nous appelons le GC majeur juste après chaque GC mineur. Les calculs du programme, du GC mineur et du GC majeur sont donc entrelacés comme le montre la figure 7.8. Cette technique a un avantage important : à la fin du GC mineur, le tas mineur est vide ainsi que la table des pointeurs inverses. Le GC majeur n'a donc pas besoin de connaître et de mettre à jour la table des pointeurs inverses (ni la table des objets en cours), contrairement à ce qui se passait pour le GC hybride (section 3.3.5).

Enfin, l'appel du GC majeur juste après le GC mineur nous permet de simplifier encore le grisage des racines au début d'un cycle de GC majeur : à la fin de son cycle, le GC majeur sait que le tas mineur est vide, donc les seules racines sont les variables globales et la pile, c'est-à-dire les racines utilisées par le GC mineur. Le GC majeur peut donc appeler la fonction d'énumération des racines (qui est fournie par le mutateur) pour obtenir toutes les racines. La même fonction d'énumération des racines sert pour le GC majeur et pour le GC mineur. On n'a donc plus besoin d'un cycle de GC mineur spécial qui copie les racines en les grisant au début du cycle de GC majeur.

Coroutines

La transformation du GC majeur en procédure incrémentale constitue la principale difficulté de l'adaptation du GC concurrent à un système incrémental. La fonction `major_gc_slice` est chargée d'effectuer une tranche de GC majeur. Elle commence par déterminer quelle est l'étape courante (*Scan* ou *Sweep*). Si l'étape courante est *Scan*, nous appelons la fonction `mark_slice`, sinon la fonction `sweep_slice`. Le passage de *Scan* à *Sweep* est fait par la fonction `mark_slice` lorsqu'elle détermine que l'étape *Scan* est terminée. De même, les étapes *Clear* et *Mark* et le début de l'étape *Scan* sont faits par la fonction `sweep_slice` à la fin de l'étape *Sweep*.

Le code de l'étape *Sweep* dans la version concurrente est constitué de deux boucles imbriquées : la boucle externe énumère les morceaux du tas et la boucle interne énumère les objets de chaque morceau. Nous ne pouvons pas utiliser cette organisation pour



A : Programme de départ avec deux boucles imbriquées.
B : On remonte `c1` à travers les deux flèches qui y mènent.
C : On remonte `i1` de la même façon.
D : Les deux flèches montantes arrivent au même endroit : on les fusionne.

Figure 7.9: Fusion de deux boucles imbriquées en une seule boucle

la version incrémentale car il faut pouvoir interrompre le balayage après avoir balayé quelques objets (le nombre exact est déterminé par la fonction `major_gc_slice` selon un algorithme expliqué ci-dessous), et il faut pouvoir reprendre le balayage là où il s'était interrompu. A cette fin, nous utilisons une technique tirée de [53], qui nous permet de remplacer deux boucles imbriquées par une seule boucle et quelques tests. Cette technique est illustrée par la figure 7.9. Le programme d'origine est :

while $condition_1$ **do**
    $instruction_1$
    while $condition_2$ **do** $instruction_2$
    $instruction_3$

Le programme transformé est le suivant (l'instruction **break** fait sortir de la boucle) :

if $condition_1$ **then**
    $instruction_1$
    while true do
        if $condition_2$ **then** $instruction_2$
        else
            $instruction_3$
            if $condition_1$ **then** $instruction_1$
            else break

On peut facilement rendre incrémentale cette nouvelle version car elle ne comporte plus qu'une boucle : on utilise un entier *work*, qui est la quantité de travail restant à accomplir dans cette tranche (sa valeur initiale est calculée par `major_gc_slice`). On remplace la condition **true** de la boucle par $work > 0$, et on ajoute une instruction qui décrémente *work*. La version incrémentale de la boucle est donc la suivante :

while $work > 0$ **do**
    if $condition_2$ **then**
        $instruction_2$
        $work \leftarrow work - 1$
    else
        $instruction_3$
        if $condition_1$ **then** $instruction_1$
        else $work \leftarrow 0$

Le même traitement a été appliqué aux quatre boucles imbriquées de l'étape *Scan*, ce qui donne un résultat à peine plus compliqué que pour les deux boucles de *Sweep*.

Épaisseur des tranches

Le calcul du paramètre `work` des fonctions `mark_slice` et `sweep_slice` est fait par la fonction `major_gc_slice`. Ce paramètre est très important car c'est lui qui règle la quantité de travail effectué par la tranche de GC majeur, la fréquence des tranches étant fixée par la fréquence des GC mineurs. Si `work` est trop grand, le GC majeur travaille

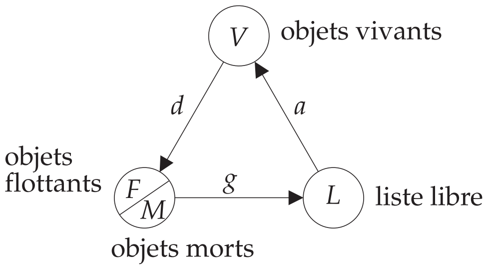


Figure 7.10: Étapes de consommation et de recyclage de la mémoire

trop vite : il consomme une quantité importante de temps de calcul pour récupérer peu de mémoire. Si `work` est trop petit, le GC majeur risque de ne pas récupérer la mémoire aussi vite qu'elle est utilisée par le mutateur ; dans ce cas, le mutateur va sans arrêt épuiser la liste libre et agrandir le tas.

Pour trouver un juste milieu, nous indexons `work` sur la quantité de mémoire allouée (dans le tas majeur) depuis la dernière tranche de GC majeur. Ainsi, un programme qui alloue vite fera plus travailler le collecteur qu'un programme qui alloue lentement.

Il reste à déterminer la constante de vitesse qui donne le rapport entre la quantité de mémoire allouée et la valeur de `work`. Pour ce faire, nous utilisons un modèle simplifié de la mémoire : nous ignorons les effets dus à la fragmentation et nous supposons que la taille totale des objets vivants est constante, c'est-à-dire que des objets deviennent inaccessibles au fur et à mesure que le mutateur en alloue de nouveaux (et à la même vitesse). Nous discuterons plus loin le cas où cette hypothèse est fausse.

Le modèle simplifié de la mémoire est illustré par la figure 7.10. Les mots de mémoire circulent entre quatre ensembles : L, V, F et M. L'ensemble L est la liste libre, V l'ensemble des objets vivants, F l'ensemble des objets flottants et M l'ensemble des objets morts (qui seront désalloués par le cycle de GC en cours). Ces ensembles sont reliés par trois flèches qui représentent le circuit effectué par la mémoire : de L, par allocation, la mémoire passe dans V puis, en devenant inaccessible, dans F.

L'étape *Sweep* du collecteur fait passer la mémoire de M à L, et le démarrage d'un cycle de GC vide d'un seul coup F dans M. Nous notons par a, d et g le “débit” des flèches, c'est-à-dire la vitesse à laquelle la mémoire passe d'un ensemble dans l'autre. a est la vitesse d'allocation, d est la vitesse de “disparition” (la vitesse à laquelle les objets deviennent inaccessibles) et g la vitesse de désallocation.

Notre unité de temps sera le mot alloué par le collecteur, c'est-à-dire que nous utilisons l'allocation comme horloge. C'est cette décision qui indexe automatiquement la vitesse du GC sur la vitesse d'allocation du mutateur. Avec cette horloge, a est égal à 1 par définition, (a vaut 1 mot alloué par mot alloué). Par hypothèse, d est égal à a, puisque la taille de V est constante (le débit entrant doit donc être égal au débit sortant).

Supposons que l'étape *Mark* prend un temps négligeable. Le GC ne désalloue les objets que pendant l'étape *Sweep*, donc g est nul pendant l'étape *Scan*. La taille de L évolue donc en dents de scie : elle croît pendant l'étape *Sweep* et diminue pendant l'étape *Scan*. Si elle tombe à zéro, la liste libre est vide et le mutateur agrandit alors le tas, augmentant ainsi brutalement la taille de L.

Supposons que le système est en régime stable, c'est-à-dire que tous les cycles de GC ont la même durée, g est constant pendant l'étape *Sweep* et la liste libre ne se vide jamais complètement. Alors la taille T du tas est constante et le débit moyen du GC doit être égal à a et à d. Appelons N la durée du GC (donc le nombre de mots alloués par le mutateur pendant un cycle de GC) et supposons que les étapes *Scan* et *Sweep* prennent le même temps : $N/2$. Dans ces conditions, on a $|F| = 0$ au début du cycle (le GC vient de vider F dans M), $|F| = N/2$ au milieu du cycle (passage de *Mark* à *Sweep*) et $|F| = N$ à la fin du cycle. On a donc $|M| = N$ au début du cycle (le GC vient de vider F, qui est de taille N, dans M), $|M| = N$ au milieu du cycle (car $g = 0$ pendant le marquage), et $|M| = 0$ à la fin du cycle. Enfin, en notant l la valeur de $|L|$ au milieu du cycle, on a $|L| = l + N/2$ au début, $|L| = l$ au milieu et $|L| = l + N/2$ à la fin du cycle.

On remarque que $|F| + |M| + |L| = l + 3N/2$ à tout moment. La valeur $3N + 2$ correspond à la quantité de mémoire qui doit se trouver dans le circuit du GC pour éviter que celui-ci ne se désamorce (c'est-à-dire pour éviter que la liste libre ne se vide complètement). Nous appellerons cette valeur le *surcoût en mémoire* du GC. La valeur l correspond à une réserve de mémoire libre qui est disponible à tout moment dans la liste libre. La situation optimale (la taille minimale du tas) correspond à $l = 0$.

L'utilisateur donne la valeur du surcoût en mémoire désiré (exprimé comme pourcentage de la taille du tas : $o = 3N/2T$) et la fonction `major_gc_slice` règle la vitesse des étapes *Mark*, *Scan* et *Sweep* en supposant que le régime stable est atteint. Elle connaît :

- Le temps disponible pour effectuer l'étape en cours ($t = N/2$).
- La quantité de travail totale de l'étape en cours ($q = T - 3N/2$ mots à marquer ou $q = T$ mots à balayer).
- La vitesse voulue ($v = q/t$).
- Le temps écoulé depuis la dernière tranche (e).

La quantité de travail à effectuer pour rattraper le temps écoulé est alors `work` $= ve$. En ne faisant pas entrer l'étape *Mark* dans le calcul de `work`, on lui donne effectivement une "vitesse" infinie (l'étape *Mark* est effectuée en plus du travail normal de la dernière tranche de *Sweep*).

Examinons le comportement du système lorsque certaines de nos hypothèses deviennent fausses :

- Si la taille de V n'est pas constante. Si elle augmente, le GC ne récupère pas autant de mémoire que prévu (car $d < a$) et la liste libre finira par se vider avant le milieu du cycle. Dans ce cas le mutateur va augmenter la taille du tas. C'est le comportement voulu : si les besoins en mémoire du programme augmentent, il faut agrandir le tas, jusqu'à atteindre la position d'équilibre. Si la taille de

programme	rendement	latence		temps de calcul
		moyenne	maximale	
KB	97 %	4,2 ms	13 ms	11,4 %
compilateur	80 %	5,0 ms	41 ms	6,3 %
CoQ	93 %	4,2 ms	89 ms	8,2 %
SIMPLE	97 %	1,1 ms	9,8 ms	3,3 %

Figure 7.11: Performances du GC incrémental

V diminue ($d > a$) alors l va augmenter (le GC récupère plus de mémoire que prévu). Il faudrait diminuer la taille du tas ou ralentir le GC. Nous ne le faisons pas pour le moment.

- Si g n'est pas constant pendant l'étape *Sweep*, alors au pire tous les objets sont désalloués juste à la fin de l'étape *Sweep*. La taille de L au début du cycle doit alors être $N+l$ (car aucun objet ne sera désalloué avant la fin du cycle, qui alloue N mots) et le surcoût en mémoire est de $2N$.
- Si on ne néglige pas la fragmentation. Nous n'avons pas d'indications quantitatives, mais remarquons que la taille de la liste libre influe sur la fragmentation : plus il y a d'objets libres dans le tas, plus il y a d'occasions de les recoller. Donc en augmentant la valeur de l on diminue la fragmentation.

Il est théoriquement possible que le tas augmente trop vite pour que le GC puisse terminer un cycle, puisque son estimation du travail à faire se base à chaque tranche sur la taille courante du tas, alors qu'il faudrait se baser sur la taille qu'aura le tas au moment où le cycle se finira. Pour éviter ce problème (et aussi pour éviter les mauvaises surprises dues à la fragmentation), nous ajoutons une petite marge de sécurité à la valeur calculée de `work`, ce qui accélère le GC et augmente donc l.

7.4.2 Performances

La figure 7.11 donne les performances du GC de Caml Light 0.7. Les programmes et les conditions d'utilisation sont les mêmes que dans la section 3.4.1. Comme le GC mineur est exactement le même, on peut comparer directement ces chiffres à ceux de la section 3.4.1.

Les temps de latence sont bien évidemment plus grands que pour le GC mineur seul, mais ils restent du même ordre de grandeur. Dans le pire des cas, le temps de pause dû au GC reste inférieur à $1/10^e$ de seconde, ce qui est parfaitement acceptable pour une application interactive. Le temps de calcul du GC majeur est de 2 à 5% du temps total du programme, ce qui est comparable avec celui du GC mineur. Cette remarque justifie l'utilisation du GC mineur : il désalloue la plupart des objets sans consommer beaucoup plus de temps de calcul que le GC majeur.

Enfin, ces chiffres ont été obtenus avec un paramètre $o = 30\%$. En augmentant o, on diminue le coût et les temps de latence du GC majeur, au prix d'un encombrement

o	latence moyenne	latence maximale	temps de calcul
10 %	9,0 ms	250 ms	16 %
20 %	5,7 ms	130 ms	11 %
30 %	4,2 ms	89 ms	8,2 %
60 %	2,8 ms	53 ms	5,5 %

Figure 7.12: Rapport entre le temps de calcul et la mémoire occupée

en mémoire plus élevé. Inversement, on peut diminuer o pour faire fonctionner le programme avec moins de mémoire, au prix d'un peu plus de temps de calcul. La figure 7.12 donne le résultat de ces manipulations sur CoQ. Si on diminue o en dessous de 10%, le temps de calcul du GC majeur devient déraisonnable.

On peut aussi augmenter le rendement du GC mineur et diminuer le temps de calcul, au prix de temps de latence plus élevés, en augmentant la taille du tas mineur. Dans Caml Light 0.7, le paramètre o et la taille du tas mineur sont contrôlés par l'utilisateur (pour leur valeur initiale) et par le programme (qui peut les changer en cours d'exécution).

Chapitre 8

Conclusion

La gestion automatique de la mémoire pose des problèmes particuliers dans le cas du langage ML, à cause de son fort taux d'allocation, et elle admet des solutions originales, grâce au faible taux de mutation et au typage spécial des objets mutables. Cette thèse montre l'intérêt d'utiliser un GC à générations dans ces conditions.

Pour les variantes de ML fonctionnant sur machines parallèles, nous avons décrit un système original de générations et son implémentation sur machines à mémoire partagée. L'efficacité de ce système est extrêmement satisfaisante.

Enfin, pour les machines parallèles à mémoire partagée, nous avons donné un algorithme prouvé, implémenté et efficace. Il s'agit à notre connaissance du premier algorithme de GC concurrent réunissant ces trois qualités. Nous avons pu constater au passage que la preuve de programme, loin d'être une considération théorique éloignée de la réalité, est une technique de déboguage remarquablement puissante.

Il reste quelques possibilités intéressantes à explorer. D'abord, l'ajout d'une ou plusieurs générations dans le GC mineur pour augmenter son rendement. Cette technique permettra sans doute d'obtenir des performances plus homogènes d'un programme à l'autre (car la durée de vie des objets varie beaucoup entre les programmes). Elle permet aussi de réduire la charge du GC majeur, donc son temps de calcul.

Ensuite, pour combattre activement la fragmentation, on peut imaginer que le GC majeur déplace les objets. Dans le cas des objets non mutables, ce déplacement peut se faire sans aucune synchronisation.

Enfin, nous envisageons d'intégrer notre système de génération locale dans un GC réparti, par exemple, celui de [64]. On peut en attendre une bonne amélioration des performances, pour un coût minime en termes de complexité du système.

Annexe A

L'algorithme de GC concurrent

Cette annexe redonne sous forme plus compacte le modèle formel de l'algorithme, les définitions auxiliaires, et les invariants de la preuve.

A.1 L'algorithme

Déclarations globales

$$
\begin{array}{ll}
\textbf{type} & \mathit{Pid} \\
& \begin{array}{ll}
\mathit{Addr} & \triangleq \mathbf{Nat} \\
\mathit{Sizes} & \triangleq \mathbf{Nat} \\
\mathit{Colors} & \triangleq \{\mathit{Blue}, \mathit{White}, \mathit{Gray}, \mathit{Black}\} \\
\mathit{Headers} & \triangleq \mathbf{record} \begin{cases} \mathit{color} \in \mathit{Colors} \\ \mathit{size} \;\; \in \mathit{Sizes} \end{cases} \\
\mathit{Words} & \triangleq \mathit{Addr} \cup \mathit{Headers} \\
\mathit{Statuses} & \triangleq \{\mathit{Async}, \mathit{Sync}_1, \mathit{Sync}_2, \mathit{Dead}, \mathit{Avail}, \mathit{Quick}\}
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{var} & \mathit{heap} & \in \mathbf{array}\ [\mathit{Addr}]\ \mathbf{of}\ \ \mathit{Words} \\
& \mathit{top} & \in \mathit{Addr} \\
& \mathit{dirty} & \in \mathit{Bool} \\
& \mathit{free}, \mathit{alloc} & \in \mathbf{multiset\ of}\ \ \mathit{Addr} \\
& \mathit{status}_C & \in \mathit{Statuses} \\
& \mathit{swept} & \in \mathit{Addr} \uplus \{-\infty, +\infty\} \\
& \mathit{scanned} & \in \mathit{Addr} \uplus \{-\infty\}
\end{array}
$$

$$
\begin{array}{l}
\forall m \in \mathit{Pid}, \\
\quad \mathit{status}_m \in \mathit{Statuses} \\
\quad \mathit{args}_m \;\; \in \mathbf{multiset\ of}\ \ \mathit{Addr}
\end{array}
$$

Initialisation

init $top = 0$
$free = alloc = \emptyset$
$status_C = Async$
$swept = -\infty$
$\forall m \in Pid, status_m \in \{Avail, Async\}$
$|\{m \in Pid \mid status_m \neq Avail\}| < \infty$
$\forall m \in Pid, args_m = \emptyset$

Mémoire

1 $\langle\ s \in Sizes$
$\Longrightarrow heap[top] \leftarrow \textbf{record} \begin{cases} color \mapsto Blue \\ size \mapsto s \end{cases}$
$alloc \leftarrow alloc \oplus \{top\}$
$top \leftarrow top + s + 1\ \rangle$
2 $\langle\ x \in free$
$\Longrightarrow heap[x].color \leftarrow Blue$
$free \leftarrow free \ominus \{x\}$
$alloc \leftarrow alloc \oplus \{x\}\ \rangle$
3 $\langle\ x \in free$
$\Longrightarrow free \leftarrow free \ominus \{x\}\ \rangle$
4 $\langle\ x \in alloc$
$\Longrightarrow alloc \leftarrow alloc \ominus \{x\}$
$heap[x].color \leftarrow White\ \rangle$

Mutateur m

type $CreateProc \triangleq \{Split, TestSweep, ClearNew, GrayNew, Fill\}$
$UpdateProc \triangleq \{TestOld, GrayOld, TestScan, SetDirty, Store\}$
$Labels \triangleq CreateProc \uplus UpdateProc \uplus \{Halt, Work, Launch\}$

var $pc = Halt \in Labels$
$roots = pool = toMark = toFill = \emptyset \in$ **multiset of** $Addr$
$answering = marking =$ **false** $\in Bool$
$child \in Pid$
$old, new, field \in Addr$

startup
5 $\langle\ pc = Halt\ \wedge\ status_m \in \{Async, Sync_1, Sync_2\}$
$\stackrel{w}{\Longrightarrow} roots \leftarrow args_m$
$args_m \leftarrow \emptyset$
$answering \leftarrow (status_m \neq status_C)$
$pc \leftarrow Work\ \rangle$

launch

6 $\langle\ pc = Work\ \wedge \neg\, answering\ \wedge p \in Pid\ \wedge\ status_p = Avail$

$\Longrightarrow child \leftarrow p$

$status_p \leftarrow Quick$

$pc \leftarrow Launch\ \rangle$

7 $\langle\ pc = Launch\ \wedge x \in roots\ \wedge p = child$

$\Longrightarrow args_p \leftarrow args_p \oplus \{x\}\ \rangle$

8 $\langle\ pc = Launch\ \wedge p = child$

$$\stackrel{w}{\Longrightarrow} status_p \leftarrow \begin{cases} Async & \textbf{if}\ marking \\ status_m & \textbf{otherwise} \end{cases}$$

$pc \leftarrow Work\ \rangle$

exit

9 $\langle\ pc = Work\ \wedge\ toMark = pool = \emptyset$

$\Longrightarrow status_m \leftarrow Dead$

$answering \leftarrow marking \leftarrow \textbf{false}$

$roots \leftarrow \emptyset$

$pc \leftarrow Halt\ \rangle$

cooperate

10 $\langle\ pc \neq Halt\ \wedge\ status_m \neq status_C$

$\stackrel{w}{\Longrightarrow} answering \leftarrow \textbf{true}\ \rangle$

11 $\langle\ pc = Work\ \wedge\ answering\ \wedge \neg\, marking$

$\stackrel{w}{\Longrightarrow} answering \leftarrow \textbf{false}$

$\textbf{if}\ status_m = Sync_2\ \textbf{then}$

$toMark \leftarrow toMark \oplus roots$

$marking \leftarrow \textbf{true}$

$$status_m \leftarrow \begin{cases} Sync_1 & \textbf{if}\ status_m = Async \\ Sync_2 & \textbf{otherwise} \end{cases}\ \rangle$$

12 $\langle\ pc = Work\ \wedge\ marking\ \wedge\ toMark = \emptyset$

$\stackrel{w}{\Longrightarrow} answering \leftarrow marking \leftarrow \textbf{false}$

$status_m \leftarrow Async\ \rangle$

mark

13 $\langle\ x \in toMark\ \wedge\ heap[x].\,color \neq White$

$\Longrightarrow toMark \leftarrow toMark \ominus \{x\}\ \rangle$

14 $\langle\ x \in toMark$

$\stackrel{w}{\Longrightarrow} heap[x].\,color \leftarrow Gray$

$toMark \leftarrow toMark \ominus \{x\}\ \rangle$

move

15 $\langle\ x \in roots$

$\Longrightarrow roots \leftarrow roots \oplus \{x\}\ \rangle$

16 $\langle\ x \in roots$

$\Longrightarrow roots \leftarrow roots \ominus \{x\}\ \rangle$

load

$$
\begin{array}{ll}
17\ \langle & x \in \mathit{roots}\ \wedge x < z \leq x + \mathit{heap}[x].\,\mathit{size} \\
& \Longrightarrow \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{heap}[z]\}\ \rangle
\end{array}
$$

reserve

$$
\begin{array}{ll}
18\ \langle & \mathit{pc} = \mathit{Work}\ \wedge x \in \mathit{alloc} \\
& \Longrightarrow \mathit{alloc} \leftarrow \mathit{alloc} \ominus \{x\} \\
& \qquad \mathit{pool} \leftarrow \mathit{pool} \oplus \{x\}\ \rangle \\
19\ \langle & x \in \mathit{pool} \\
& \Longrightarrow \mathit{pool} \leftarrow \mathit{pool} \ominus \{x\} \\
& \qquad \mathit{alloc} \leftarrow \mathit{alloc} \oplus \{x\}\ \rangle
\end{array}
$$

create

$$
\begin{array}{ll}
20\ \langle & \mathit{pc} = \mathit{Work}\ \wedge \neg\, \mathit{answering}\ \wedge s \in \mathit{Sizes}\ \wedge x \in \mathit{pool}\ \wedge s \leq \mathit{heap}[x].\,\mathit{size} \\
& \Longrightarrow \mathit{pool} \leftarrow \mathit{pool} \ominus \{x\} \\
& \qquad \mathit{old} \leftarrow x \\
& \qquad \mathit{new} \leftarrow x + \mathit{heap}[x].\,\mathit{size} - s \\
& \qquad \mathit{toFill} \leftarrow \{\mathit{new} + 1, \ldots, \mathit{new} + s\} \\
& \qquad \mathit{heap}[\mathit{new}] \leftarrow \textbf{record} \begin{cases} \mathit{color} \mapsto \mathit{Black} \\ \mathit{size} \ \ \mapsto s \end{cases} \\
& \qquad \mathit{pc} \leftarrow \begin{cases} \mathit{Split} & \textbf{if}\ \ \mathit{old} < \mathit{new} \\ \mathit{TestSweep} & \textbf{otherwise}\ \ \rangle \end{cases} \\
21\ \langle & \mathit{pc} = \mathit{Split} \\
& \overset{w}{\Longrightarrow} \mathit{heap}[\mathit{old}] \leftarrow \ \textbf{record} \begin{cases} \mathit{color} \mapsto \mathit{Blue} \\ \mathit{size} \ \ \mapsto \mathit{new} - \mathit{old} - 1 \end{cases} \\
& \qquad \mathit{pool} \leftarrow \mathit{pool} \oplus \{\mathit{old}\} \\
& \qquad \mathit{pc} \leftarrow \mathit{TestSweep}\ \rangle \\
22\ \langle & \mathit{pc} = \mathit{TestSweep} \\
& \overset{w}{\Longrightarrow} \mathit{pc} \leftarrow \begin{cases} \mathit{ClearNew} & \textbf{if}\ \ \mathit{status}_m \neq \mathit{Async}\ \vee\ \mathit{new} < \mathit{swept} \\ \mathit{GrayNew} & \textbf{if}\ \ \mathit{status}_m = \mathit{Async}\ \wedge\ \mathit{old} \leq \mathit{swept} \leq \mathit{new} \\ \mathit{Fill} & \textbf{otherwise}\ \ \rangle \end{cases} \\
23\ \langle & \mathit{pc} = \mathit{ClearNew} \\
& \overset{w}{\Longrightarrow} \mathit{heap}[\mathit{new}].\,\mathit{color} \leftarrow \mathit{White} \\
& \qquad \mathit{pc} \leftarrow \mathit{Fill}\ \rangle \\
24\ \langle & \mathit{pc} = \mathit{GrayNew} \\
& \overset{w}{\Longrightarrow} \mathit{heap}[\mathit{new}].\,\mathit{color} \leftarrow \mathit{Gray} \\
& \qquad \mathit{pc} \leftarrow \mathit{Fill}\ \rangle
\end{array}
$$

fill

$$
\begin{array}{ll}
25\ \langle & y \in \mathit{roots} \cup \{\mathit{new}\}\ \wedge\ z \in \mathit{toFill} \\
& \overset{w}{\Longrightarrow} \mathit{heap}[z] \leftarrow y \\
& \qquad \textbf{if}\ \ \mathit{marking}\ \ \textbf{then}\ \ \mathit{toMark} \leftarrow \mathit{toMark} \ominus \{y\} \\
& \qquad \mathit{toFill} \leftarrow \mathit{toFill} \ominus \{z\}\ \rangle \\
26\ \langle & \mathit{pc} = \mathit{Fill}\ \wedge\ \mathit{toFill} = \emptyset \\
& \overset{w}{\Longrightarrow} \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{new}\} \\
& \qquad \textbf{if}\ \ \mathit{marking}\ \ \textbf{then}\ \ \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{new}\} \\
& \qquad \mathit{pc} \leftarrow \mathit{Work}\ \rangle
\end{array}
$$

update

27 $\langle\ pc = \mathit{Work}\ \wedge \neg\, \mathit{answering}\ \wedge x, y \in \mathit{roots}\ \wedge x < z \leq x + \mathit{heap}[x].\,\mathit{size}$

$$\Longrightarrow \mathit{new} \leftarrow y$$
$$\mathit{field} \leftarrow z$$
$$\mathit{old} \leftarrow \mathit{heap}[z]$$
$$\textbf{if } \mathit{status}_m \neq \mathit{Async}\ \wedge \neg\, \mathit{marking}\ \textbf{ then } \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{new}\}$$
$$\textbf{if } \mathit{status}_m = \mathit{Sync}_2\ \textbf{ then } \mathit{toMark} \leftarrow \mathit{toMark} \oplus \{\mathit{old}\}$$
$$pc \leftarrow \begin{cases} \mathit{TestOld} & \textbf{if } \mathit{status}_m = \mathit{Async} \\ \mathit{Store} & \textbf{otherwise}\ \rangle \end{cases}$$

28 $\langle\ pc \in \mathit{UpdateProc}\ \wedge\ \mathit{swept} > -\infty$

$$\Longrightarrow pc \leftarrow \mathit{Store}\ \rangle$$

29 $\langle\ pc = \mathit{TestOld}$

$$\stackrel{w}{\Longrightarrow} pc \leftarrow \begin{cases} \mathit{GrayOld} & \textbf{if } \mathit{heap}[\mathit{old}].\,\mathit{color} = \mathit{White} \\ \mathit{TestScan} & \textbf{if } \mathit{heap}[\mathit{old}].\,\mathit{color} = \mathit{Gray} \\ \mathit{Store} & \textbf{otherwise}\ \rangle \end{cases}$$

30 $\langle\ pc = \mathit{GrayOld}$

$$\stackrel{w}{\Longrightarrow} \mathit{heap}[\mathit{old}].\,\mathit{color} \leftarrow \mathit{Gray}$$
$$pc \leftarrow \mathit{TestScan}\ \rangle$$

31 $\langle\ pc = \mathit{TestScan}$

$$\stackrel{w}{\Longrightarrow} pc \leftarrow \begin{cases} \mathit{SetDirty} & \textbf{if } \mathit{old} \leq \mathit{scanned} \\ \mathit{Store} & \textbf{otherwise}\ \rangle \end{cases}$$

32 $\langle\ pc = \mathit{SetDirty}$

$$\stackrel{w}{\Longrightarrow} \mathit{dirty} \leftarrow \textbf{true}$$
$$pc \leftarrow \mathit{Store}\ \rangle$$

33 $\langle\ pc = \mathit{Store}$

$$\stackrel{w}{\Longrightarrow} \mathit{heap}[\mathit{field}] \leftarrow \mathit{new}$$
$$pc \leftarrow \mathit{Work}\ \rangle$$

Collecteur

type $\mathit{Steps} \triangleq \{\mathit{Sweep}, \mathit{Clear}, \mathit{Mark}, \mathit{Scan}\}$

var $\mathit{step} = \mathit{Sweep} \in \mathit{Steps}$
$\mathit{phase} = \mathit{Async} \in \mathit{Statuses}$
$\mathit{ptr} = \mathit{limit} = \mathit{sublimit} = \mathit{rover} = 0 \in \mathit{Addr}$
$\mathit{reset} = \textbf{true} \in \mathit{Bool}$
$\mathit{toWhite} = \mathit{toBlack} = \mathit{toTrace} = \emptyset \in \textbf{set of } \mathit{Addr}$
$\mathit{claim} = \mathit{cache} = \mathit{fields} = \emptyset \in \textbf{multiset of } \mathit{Addr}$

sweep

34 $\langle\ \mathit{step} = \mathit{Sweep}\ \wedge\ \mathit{swept} = \mathit{ptr} < \mathit{sublimit}$

$$\stackrel{w}{\Longrightarrow} \textbf{if } \mathit{heap}[\mathit{ptr}].\,\mathit{color} \in \{\mathit{Gray}, \mathit{Black}\}\ \textbf{ then } \mathit{toWhite} \leftarrow \mathit{toWhite} \cup \{\mathit{ptr}\}$$
$$\textbf{else if } \mathit{heap}[\mathit{ptr}].\,\mathit{color} = \mathit{White}\ \textbf{ then } \mathit{claim} \leftarrow \mathit{claim} \oplus \{\mathit{ptr}\}$$
$$\mathit{ptr} \leftarrow \mathit{ptr} + \mathit{heap}[\mathit{ptr}].\,\mathit{size} + 1\ \rangle$$

35 $\langle\ \mathit{step} = \mathit{Sweep} \ \wedge\ \mathit{swept} < \mathit{ptr}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{swept} \leftarrow \mathit{ptr}\ \rangle$

36 $\langle\ \mathit{step} = \mathit{Sweep} \ \wedge\ \mathit{ptr} < x \leq \mathit{limit}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{free} \leftarrow \mathit{free} \ominus \{\mathit{ptr}, \ldots, x-1\}$
$\qquad\quad \mathit{sublimit} \leftarrow x\ \rangle$

37 $\langle\ \mathit{step} = \mathit{Sweep} \ \wedge\ \mathit{ptr} = \mathit{limit}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{swept} \leftarrow +\infty\ \rangle$

38 $\langle\ \mathit{step} = \mathit{Sweep} \ \wedge\ \mathit{swept} = +\infty$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{limit} \leftarrow \mathit{top}$
$\qquad\quad \mathit{step} \leftarrow \mathit{Clear}\ \rangle$

clear

39 $\langle\ \mathit{step} = \mathit{Clear} \ \wedge\ \mathit{ptr} < \mathit{limit}$
$\qquad \stackrel{w}{\Longrightarrow} \mathbf{if}\ \ \mathit{heap}[\mathit{ptr}].\mathit{color} \in \{\mathit{Gray}, \mathit{Black}\}\ \mathbf{then}\ \ \mathit{toWhite} \leftarrow \mathit{toWhite} \cup \{\mathit{ptr}\}$
$\qquad\quad \mathit{ptr} \leftarrow \mathit{ptr} + \mathit{heap}[\mathit{ptr}].\mathit{size} + 1\ \rangle$

40 $\langle\ x \in \mathit{toWhite}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{toWhite} \leftarrow \mathit{toWhite} \setminus \{x\}$
$\qquad\quad \mathit{heap}[x].\mathit{color} \leftarrow \mathit{White}\ \rangle$

41 $\langle\ \mathit{step} = \mathit{Clear} \ \wedge\ \mathit{ptr} = \mathit{limit} \ \wedge\ \mathit{toWhite} = \emptyset$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{status}_C \leftarrow \mathit{Sync}_1\ \rangle$

42 $\langle\ \mathit{step} = \mathit{Clear} \ \wedge\ \mathit{phase} = \mathit{Sync}_1 \ \wedge\ \mathit{claim} = \emptyset$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{status}_C \leftarrow \mathit{Sync}_2$
$\qquad\quad \mathit{step} \leftarrow \mathit{Mark}\ \rangle$

claim

43 $\langle\ x \in \mathit{claim} \ \wedge y = x + \mathit{heap}[x].\mathit{size} + 1 \in \mathit{claim}$
$\qquad \Longrightarrow \mathit{claim} \leftarrow \mathit{claim} \ominus \{y\}$
$\qquad\quad \mathit{heap}[x] \leftarrow\ \mathbf{record} \begin{cases} \mathit{color} \mapsto \mathit{White} \\ \mathit{size} \ \ \mapsto \mathit{heap}[x].\mathit{size} + \mathit{heap}[y].\mathit{size} + 1\ \rangle \end{cases}$

44 $\langle\ x \in \mathit{claim}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{claim} \leftarrow \mathit{claim} \ominus \{x\}$
$\qquad\quad \mathit{free} \leftarrow \mathit{free} \oplus \{x\}\ \rangle$

handshake

45 $\langle\ \mathit{status}_C \neq \mathit{phase} \ \wedge \forall m \in \mathit{Pid}, \mathit{status}_m \neq \mathit{phase}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{phase} \leftarrow \mathit{status}_C\ \rangle$

46 $\langle\ \mathit{status}_m = \mathit{Dead}$
$\qquad \stackrel{w}{\Longrightarrow} \mathit{status}_m \leftarrow \mathit{Avail}\ \rangle$

globals

47 $\langle\ \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\}$
$\qquad \Longrightarrow \mathit{rover} \leftarrow 0\ \rangle$

48 $\langle\ \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\} \ \wedge\ \mathit{rover} < \mathit{top}$
$\qquad \Longrightarrow \mathit{rover} \leftarrow \mathit{rover} + \mathit{heap}[\mathit{rover}].\mathit{size} + 1\ \rangle$

49 $\langle\ \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\} \ \wedge\ \mathit{rover} < \mathit{top} \ \wedge\ \mathit{heap}[\mathit{rover}].\mathit{color} = \mathit{Gray}$
$\qquad \Longrightarrow \mathit{toBlack} \leftarrow \mathit{toBlack} \cup \{\mathit{rover}\}\ \rangle$

trace

50 $\langle\ x \in \mathit{toBlack}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{heap}[x].\,\mathit{color} \leftarrow \mathit{Black}$
$\qquad \mathit{toBlack} \leftarrow \mathit{toBlack} \setminus \{x\}$
$\qquad \mathit{cache} \leftarrow \mathit{cache} \oplus \{x\}\ \rangle$

51 $\langle\ x \in \mathit{cache}\ \wedge\ \mathit{cache} \neq \{x\}$
$\quad\Longrightarrow \mathit{cache} \leftarrow \mathit{cache} \ominus \{x\}$
$\qquad \mathit{heap}[x].\,\mathit{color} \leftarrow \mathit{Gray}$
$\qquad$ **if** $x < \mathit{ptr}$ **then** $\mathit{reset} \leftarrow$ **true** $\rangle$

52 $\langle\ x \in \mathit{cache}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{cache} \leftarrow \mathit{cache} \ominus \{x\}$
$\qquad \mathit{fields} \leftarrow \mathit{fields} \oplus \{x+1, \ldots, x + \mathit{heap}[x].\,\mathit{size}\}\ \rangle$

53 $\langle\ x \in \mathit{fields}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{fields} \leftarrow \mathit{fields} \ominus \{x\}$
$\qquad \mathit{toTrace} \leftarrow \mathit{toTrace} \cup \{\mathit{heap}[x]\}\ \rangle$

54 $\langle\ x \in \mathit{toTrace}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{toTrace} \leftarrow \mathit{toTrace} \setminus \{x\}$
$\qquad$ **if** $\mathit{heap}[x].\,\mathit{color} \in \{\mathit{White}, \mathit{Gray}\}$ **then** $\mathit{toBlack} \leftarrow \mathit{toBlack} \cup \{x\}\ \rangle$

mark

55 $\langle\ \mathit{phase} \neq \mathit{Async}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{swept} \leftarrow -\infty\ \rangle$

56 $\langle\ \mathit{step} = \mathit{Mark}\ \wedge\ \mathit{phase} = \mathit{Sync}_2\ \wedge\ \mathit{swept} = -\infty$
$\quad\stackrel{w}{\Longrightarrow} \mathit{status}_C \leftarrow \mathit{Async}\ \rangle$

57 $\langle\ \mathit{step} = \mathit{Mark}\ \wedge\ \mathit{phase} = \mathit{Async}$
$\quad\stackrel{w}{\Longrightarrow} \mathit{ptr} \leftarrow \mathit{limit} \leftarrow \mathit{top}$
$\qquad \mathit{step} \leftarrow \mathit{Scan}\ \rangle$

reset

58 $\langle\ \mathit{reset}\ \vee\ \mathit{ptr} = \mathit{limit}\ \wedge\ \mathit{cache} = \mathit{fields} = \mathit{toBlack} = \mathit{toTrace} = \emptyset$
$\quad\stackrel{w}{\Longrightarrow}$ **if** $\mathit{step} = \mathit{Scan}$ **then** $\mathit{ptr} \leftarrow \mathit{limit}$
$\qquad \mathit{scanned} \leftarrow -\infty\ \rangle$

59 $\langle\ \mathit{step} = \mathit{Scan}\ \wedge\ \mathit{reset}\ \wedge\ \mathit{scanned} = -\infty$
$\quad\stackrel{w}{\Longrightarrow} \mathit{ptr} \leftarrow 0$
$\qquad \mathit{reset} \leftarrow \mathit{dirty} \leftarrow$ **false** $\rangle$

60 $\langle\ \mathit{step} = \mathit{Scan}\ \wedge\ \mathit{dirty}\ \wedge (\mathit{scanned} < \mathit{ptr}\ \vee\ \mathit{ptr} = \mathit{limit})$
$\quad\stackrel{w}{\Longrightarrow} \mathit{reset} \leftarrow$ **true** $\rangle$

scan

61 $\langle\ \mathit{step} = \mathit{Scan}\ \wedge\ \mathit{scanned} < \mathit{ptr}\ \wedge \neg(\mathit{reset}\ \wedge\ \mathit{scanned} = -\infty)$
$\quad\stackrel{w}{\Longrightarrow} \mathit{scanned} \leftarrow \mathit{ptr}\ \rangle$

62 $\langle\ \mathit{step} = \mathit{Scan}\ \wedge\ \mathit{scanned} = \mathit{ptr} < \mathit{limit}$
$\quad\stackrel{w}{\Longrightarrow}$ **if** $\mathit{heap}[\mathit{ptr}].\,\mathit{color} = \mathit{Gray}$ **then** $\mathit{toBlack} \leftarrow \mathit{toBlack} \cup \{\mathit{ptr}\}$
$\qquad \mathit{ptr} \leftarrow \mathit{ptr} + \mathit{heap}[\mathit{ptr}].\,\mathit{size} + 1\ \rangle$

$$
\begin{array}{l}
63 \ \langle \ step = Scan \ \wedge \ ptr = limit \ \wedge \neg \, reset \ \wedge \neg \, dirty \\
\qquad \wedge \ cache = fields = toBlack = toTrace = \emptyset \\
\qquad\quad \overset{w}{\Longrightarrow} reset \leftarrow \mathbf{true} \\
\qquad\qquad\quad rover \leftarrow ptr \leftarrow sublimit \leftarrow 0 \\
\qquad\qquad\quad step \leftarrow Sweep \ \rangle
\end{array}
$$

A.2 Définitions auxiliaires

$$\begin{array}{ll}
\mathsf{Af}(x) & \triangleq x + heap[x].\,size + 1 \\
x\mathsf{A}y & \triangleq x < top \ \wedge \ heap[x] \in Headers \ \wedge y = \mathsf{Af}(x) \\
\mathsf{Obj} & \triangleq \mathsf{A}^*(0) \cap [0, top[\\
\mathsf{Wh} & \triangleq \{x \in \mathsf{Obj} |\ heap[x].\,color = White\} \\
\mathsf{Gr} & \triangleq \{x \in \mathsf{Obj} |\ heap[x].\,color = Gray\} \\
\mathsf{Bk} & \triangleq \{x \in \mathsf{Obj} |\ heap[x].\,color = Black\} \\
\mathsf{Bu} & \triangleq \{x \in \mathsf{Obj} |\ heap[x].\,color = Blue\} \\
\mathsf{FreeList}_m & \triangleq pool_m \oplus \begin{cases} \{old_m\} & \textbf{if } \ pc_m = Split \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{Unswept} & \triangleq \begin{cases} [ptr, limit[& \textbf{if } \ step = Sweep \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{Fr} & \triangleq (\mathsf{Wh} \cap \mathsf{Unswept}) \cup claim \cup free \cup \mathsf{Bu} \\
\mathsf{Creating}_m & \triangleq \begin{cases} \{new_m\} & \textbf{if } \ pc_m \in CreateProc \setminus \{Split\} \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{Val} & \triangleq \mathsf{Obj} \setminus (\mathsf{Fr} \cup \bigcup \mathsf{Creating}_{Pid}) \\
\mathsf{Cf}(x) & \triangleq]x, \mathsf{Af}(x)[\\
x\mathsf{C}y & \triangleq x \in \mathsf{Val} \ \wedge \ y \in \mathsf{Cf}(x) \\
x\mathsf{Hp}y & \triangleq x \in \mathsf{C}(\mathsf{Val}) \ \wedge \ y = heap[x] \\
x\mathsf{P}y & \triangleq x \in \mathsf{Val} \ \wedge \ y \in \mathsf{Hp}(\mathsf{C}(x)) \\
\mathsf{AccField}_m & \triangleq \{heap[x] |\ pc_m \in CreateProc \ \wedge x \in \mathsf{Cf}(new_m) \setminus toFill_m\} \setminus \{new_m\} \\
\mathsf{AccUpd}_m & \triangleq \begin{cases} \{new_m, old_m\} \cup \{x \in \mathsf{Val} | x\mathsf{C}\, field_m\} & \textbf{if } \ pc_m \in UpdateProc \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{AccArg}_m & \triangleq \begin{cases} \widehat{args}_{child_m} & \textbf{if } \ pc_m = Launch \\ \emptyset & \textbf{if } \ pc_m \neq Launch \ \wedge \ status_m = Quick \\ \widehat{args}_m & \textbf{otherwise} \end{cases} \\
\mathsf{Acc}_m & \triangleq \widehat{roots}_m \cup \widehat{toMark}_m \cup \mathsf{AccArg}_m \cup \mathsf{AccField}_m \cup \mathsf{AccUpd}_m \\
\mathsf{Kill}_m & \triangleq \begin{cases} \{field_m\} & \textbf{if } \ \wedge \ pc_m \in UpdateProc \ \wedge \ status_m \neq Sync_1 \\ & \qquad \wedge \ (old_m \in \mathsf{Hp}(field_m) \ \Rightarrow \ pc_m = Store) \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{Done} & \triangleq \mathsf{Bk} \setminus \widehat{cache} \\
\mathsf{FDone} & \triangleq \mathsf{C}(\mathsf{Done} \cap \mathsf{Val}) \setminus \widehat{fields}
\end{array}$$

$$x\,\mathsf{Tr}\,y \triangleq x \in \mathsf{Val}\backslash\mathsf{Done} \;\wedge\; y \in \mathsf{Hp}(\mathsf{Cf}(x)\backslash \bigcup \mathsf{Kill}_{Pid})$$

$$\mathsf{ReachMark}(m) \triangleq \widehat{toMark}_m \cup \begin{cases} \mathsf{AccField}_m & \textbf{if } marking_m \\ \emptyset & \textbf{otherwise} \end{cases}$$

$$\mathsf{ReachWh} \triangleq \mathsf{ReachMark}(Pid) \cup toBlack \cup toTrace$$

$$\mathsf{ReachFields} \triangleq \widehat{fields}\backslash \bigcup \mathsf{Kill}_{Pid}$$

$$\mathsf{ScanDone} \triangleq \begin{cases} \emptyset & \textbf{if } reset \vee dirty \\ [0, ptr[& \textbf{otherwise} \end{cases}$$

$$\mathsf{Reach} \triangleq \mathsf{ReachWh} \cup (\mathsf{Gr}\backslash\mathsf{ScanDone}) \cup \widehat{cache} \cup \mathsf{Hp}(\mathsf{ReachFields})$$

$$\mathsf{Mrk} \triangleq \mathsf{Tr}^*(\mathsf{Reach}) \cup (\mathsf{Val}\backslash\mathsf{Wh})$$

$$\mathsf{Q} \triangleq \{p \in Pid \mid status_p = Quick\}$$

$$\mathsf{L}_m \triangleq \begin{cases} \{child_m\} & \textbf{if } pc_m = Launch \\ \emptyset & \textbf{otherwise} \end{cases}$$

$$\mathsf{N}_m \triangleq \begin{cases} \{new_m\} & \textbf{if } pc_m \in \{TestSweep, ClearNew, GrayNew\} \\ \emptyset & \textbf{otherwise} \end{cases}$$

A.3 Les invariants

$I_1 \triangleq heap \in \mathbf{array}\ [Addr]\ \mathbf{of}\ \ Words \ \wedge\ top \in \mathsf{A}^*(0)$

$I_2 \triangleq alloc \oplus \bigoplus \mathsf{FreeList}_{Pid} = \mathsf{Bu}$

$I_3 \triangleq free \oplus claim \subset \mathsf{Wh}$

$I_4 \triangleq \mathsf{P}^*(\bigcup \mathsf{Acc}_{Pid}) \subset \mathsf{Val} \ \wedge\ \bigoplus \mathsf{Creating}_{Pid} \subset \mathsf{Obj}\backslash\mathsf{Fr}$

$I_5 \triangleq step \in Steps \ \wedge\ phase \in \{Async, Sync_1, Sync_2\}$

$$\begin{aligned}
I_6 \triangleq\ & \vee\ status_C = phase = Async \\
& \vee\ status_C = Sync_1 \ \wedge\ phase \neq Sync_2 \ \wedge\ step = Clear \\
& \vee\ status_C = Sync_2 \ \wedge\ phase \neq Async \ \wedge\ step = Mark \\
& \vee\ status_C = Async \ \wedge\ phase = Sync_2 \ \wedge\ step = Mark
\end{aligned}$$

$$\begin{aligned}
I_7 \triangleq\ & \wedge\ limit \in \mathsf{Obj} \cup \{top\} \\
& \wedge\ ptr \in \mathsf{Obj} \cup \{top\} \ \wedge\ ptr \leq limit \\
& \wedge\ sublimit \in Addr \ \wedge\ sublimit \leq limit
\end{aligned}$$

$I_8 \triangleq claim \subset \mathsf{Wh} \cap [0, ptr[\ \wedge\ (step \in \{Mark, Scan\} \ \Rightarrow\ claim = \emptyset)$

$I_9 \triangleq toWhite \subset (\mathsf{Obj}\backslash\mathsf{Fr}) \cap [0, ptr[\ \wedge\ (step \in \{Mark, Scan\} \ \Rightarrow\ toWhite = \emptyset)$

$$\begin{aligned}
I_{10} \triangleq\ & \vee\ swept = -\infty \ \wedge\ (step = Clear \ \Rightarrow\ phase \neq Async) \\
& \vee\ swept \in Addr \ \wedge\ step = Sweep \ \wedge\ swept \leq ptr \\
& \vee\ swept = +\infty \ \wedge\ step \neq Scan \\
& \qquad\qquad\qquad\ \wedge\ (step = Mark \ \Rightarrow\ status_C \neq Async) \\
& \qquad\qquad\qquad\ \wedge\ (step = Sweep \ \Rightarrow\ ptr = limit)
\end{aligned}$$

$I_{11} \triangleq step = Sweep \ \Rightarrow\ \mathsf{Bk} \subset toWhite \cup \bigcup \mathsf{N}_{Pid} \cup [ptr, top[$

$I_{12} \triangleq step = Sweep \ \Rightarrow\ free \cap [ptr, sublimit[= \emptyset$

$I_{13} \triangleq step = Sweep \ \Rightarrow\ (toWhite \cup \mathsf{Wh} \cup \mathsf{Bu}) \cap]swept, ptr[= \emptyset$

$I_{14} \triangleq step = Clear \ \Rightarrow\ \mathsf{Bk} \subset toWhite \cup \bigcup \mathsf{N}_{Pid} \cup [ptr, limit[$

$I_{15} \triangleq step = Clear \ \wedge\ status_C = Sync_1 \ \Rightarrow\ ptr = limit \ \wedge\ toWhite = \emptyset$

$I_{16} \triangleq rover \in \mathsf{Obj} \cup \{top\} \ \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ rover = 0)$

$I_{17} \triangleq toBlack \subset \mathsf{Val}\backslash\mathsf{Bk} \ \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ toBlack = \emptyset)$

$I_{18} \triangleq toTrace \subset \mathsf{Val} \ \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ toTrace = \emptyset)$

$I_{19} \triangleq cache \in \mathbf{multiset\ of}\ \mathsf{Val}\backslash\mathsf{Wh} \ \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ cache = \emptyset)$

$I_{20} \triangleq fields \in \mathbf{multiset\ of}\ \mathsf{C}(\mathsf{Val}\backslash\mathsf{Wh}) \ \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ fields = \emptyset)$

$$\begin{aligned}
I_{21} \triangleq\ & \wedge\ (step \in \{Mark, Scan\} \\
& \qquad \Rightarrow\ \wedge\ \mathsf{Hp}(\bigcup \mathsf{Kill}_{Pid}) \cup \mathsf{P}(\mathsf{Mrk}) \subset \mathsf{Mrk} \subset \mathsf{Val} \\
& \qquad\quad\ \ \wedge\ (status_C \neq Async \ \Rightarrow\ \mathsf{Wh} \cap \mathsf{Hp}(\mathsf{FDone} \cup \bigcup \mathsf{Kill}_{Pid}) \subset \mathsf{ReachWh})) \\
& \wedge\ (step \notin \{Mark, Scan\} \ \Rightarrow\ \mathsf{P}^*(\mathsf{Gr} \cap \mathsf{Val}) \subset \mathsf{Val})
\end{aligned}$$

$I_{22} \triangleq scanned \in Addr \cup \{-\infty\}$

$I_{23} \triangleq \neg\, reset \;\Rightarrow\; step = Scan \,\wedge\, scanned \leq ptr$

$I_{24} \triangleq step = Scan \;\Rightarrow\; \mathsf{Mrk}\backslash\mathsf{Done} \subset [0, limit[$

$$I_{25} \triangleq step = Scan \,\wedge \neg\, dirty \,\wedge \neg\, reset \,\wedge \mathsf{Mrk} \cap \mathsf{Wh} \neq \emptyset \\ \Rightarrow\;]scanned, ptr[\cap \mathsf{Obj} \subset \mathsf{Done}$$

$I_{26} \triangleq \forall m \in Pid, status_m \in \{status_C, phase, Dead, Avail, Quick\}$

$I_{27} \triangleq \bigoplus \mathsf{L}_{Pid} = \mathsf{Q}$

$I_{28} \triangleq \forall m \in Pid, args_m \in \textbf{multiset of } Addr$

$I_{29} \triangleq \forall m \in Pid, answering_m \,\vee\, marking_m \;\Rightarrow\; pc_m \neq Halt \,\wedge\, status_m \neq status_C$

$$I_{30} \triangleq \forall m \in Pid, \;\wedge\; (marking_m \;\Rightarrow\; status_C = Async) \\ \wedge\; (status_m = Async \;\Rightarrow\; toMark_m = \emptyset)$$

$$I_{31} \triangleq \forall m \in Pid, (status_m = Async \,\vee\, marking_m) \,\wedge\, step \in \{Mark, Scan\} \\ \Rightarrow\; \mathsf{Acc}_m \subset \mathsf{Mrk}$$

$I_{32} \triangleq \forall m \in Pid, pc_m \in Labels \,\wedge\, (status_m \in \{Dead, Avail, Quick\} \;\Rightarrow\; pc_m = Halt)$

$I_{33} \triangleq \forall m \in Pid, pc_m = Halt \;\Rightarrow\; roots_m = toMark_m = pool_m = \emptyset$

$I_{34} \triangleq \forall m \in Pid, pc_m \neq Halt \,\vee\, status_m \in \{Dead, Avail\} \;\Rightarrow\; args_m = \emptyset$

$$I_{35} \triangleq \forall m \in Pid, \;\wedge\; (pc_m \in CreateProc \;\Rightarrow\; toFill_m \subset \mathsf{Cf}(new_m) \,\wedge\, old_m \in Addr) \\ \wedge\; (pc_m \notin CreateProc \;\Rightarrow\; toFill_m = \emptyset)$$

$$I_{36} \triangleq \forall m \in Pid, pc_m = Split \;\Rightarrow\; \wedge\; new_m \in \mathsf{Cf}(old_m) \,\wedge\, \mathsf{Af}(new_m) = \mathsf{Af}(old_m) \\ \wedge\; heap[new_m] \in Headers \\ \wedge\; heap[new_m].\, color = Black$$

$$I_{37} \triangleq \forall m \in Pid, \;\wedge\; pc_m = TestSweep \,\wedge\, step = Sweep \\ \wedge\; swept < old_m \,\wedge\, new_m \in [swept, ptr[\\ \Rightarrow\; new_m \in \mathsf{Wh} \cup toWhite$$

$$I_{38} \triangleq \forall m \in Pid, step \in \{Mark, Scan\} \,\vee\, status_m \neq Async \\ \Rightarrow\; \mathsf{Creating}_m \subset \mathsf{Wh} \cup \mathsf{Bk} \,\wedge\, pc_m \neq GrayNew$$

$$I_{39} \triangleq \forall m \in Pid, step \in \{Mark, Scan\} \,\wedge\, status_m = Async \\ \Rightarrow\; \mathsf{Creating}_m \subset \mathsf{Bk} \,\wedge\, pc_m \neq ClearNew$$

$I_{40} \triangleq \forall m \in Pid, pc_m = ClearNew \,\wedge\, step = Sweep \;\Rightarrow\; new_m < swept$

$I_{41} \triangleq \forall m \in Pid, pc_m = Fill \,\wedge\, status_m \neq Async \;\Rightarrow\; new_m \in \mathsf{Wh}$

$I_{42} \triangleq \forall m \in Pid, pc_m \in UpdateProc \;\Rightarrow\; field_m \in \mathsf{C}(\mathsf{Val})$

$I_{43} \triangleq \forall m \in Pid, pc_m \in UpdateProc \backslash \{Store\} \;\Rightarrow\; status_m = Async$

$$
\begin{aligned}
I_{44} \triangleq \forall m \in Pid, pc_m = GrayOld \ \wedge\ step = Scan \\
&\Rightarrow\ \wedge\ old_m < limit \\
&\ \wedge\ (old_m \in]scanned, ptr[\ \Rightarrow\ dirty\ \vee\ reset\ \vee \mathsf{Mrk} \cap \mathsf{Wh} = \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
I_{45} \triangleq \forall m \in Pid, pc_m \in \{TestScan, SetDirty\}\ \wedge\ step \in \{Mark, Scan\} \\
&\Rightarrow\ old_m \in \mathsf{Bk} \cup \mathsf{Gr}
\end{aligned}
$$

$$
\begin{aligned}
I_{46} \triangleq \forall m \in Pid, pc_m = Store\ \wedge\ status_m \neq Async\ \wedge \neg\, marking_m \\
&\Rightarrow\ new_m \in \widehat{toMark}_m \cup (\mathsf{Val} \backslash \mathsf{Wh})
\end{aligned}
$$

Annexe B

```
Auteur: Georges Gonthier
Date: Tue Mar  7 14:00:20 1995
Sujet: All conjectures have been proved.


    Apres quelque temps d'effort, toul a fini par venir a bout de
notre preuve de GC concurrent. On peut donc maintenant se vanter
d'avoir le premier algo de GC concurrent realiste et PROUVE.
   Quelques statistiques :
 Longueur du pseudo-code de l'algo:     126 lignes (dont 40 de declarations)
 Longueur du modele formel:             268 lignes (63 actions)
 Longueur du modele TLP:                791 lignes
 Longueur de l'invariant:               118 lignes (33 definitions en 46 lignes
                                                    46 clauses en 72 lignes)
 Longueur de l'invariant TLP:           247 lignes
 Longueur du script TLP:              22712 lignes (y compris le modele,
                                                    l'invariant,
                                                    et 2032 lignes d'axiomes
                                                    et de lemmes sur les
                                                    ensembles et relations)
 Longueur du LP genere:              126991 lignes (par le prepro TLP)
 Longueur du log LP:                 606483 lignes (ne trace que les deductions
                                                    pas les recritures)
 Temps CPU pris par LP:            23:50:28 (presque une journee!)
 Temps reel (LP+swap+TLP+emacs):        ~33 heures
 Taille de l'image memoire:        Out of memory!

Le tout sur une pauvre DecStation 5000/200 a 64 Mo, et passe en trois
morceaux parce que LP 2.4 sature quand son tas depasse les 50 Mo.

  Maintenant, je serais curieux de voir ce que ca peut donne dans un
autre prouveur...

  Georges
```

Références

[1] Martín Abadi — *An Axiomatization of Lamport's Temporal Logic of Actions* — Rapport de recherche n° 65, octobre 1990, DEC Systems Research Center, Palo Alto, California. Revu en mars 1993. Autre version: [2].

[2] Martín Abadi — An axiomatization of Lamport's temporal logic of actions. *In: Theories of Concurrency: Unification and Extension (CONCUR)*, éd. par J. C. M. Baeten et J. W. Klop. Lecture Notes in Computer Science (LNCS) n° 458, pp. 57–69 — 1990, Springer-Verlag. Autre version de [1].

[3] Saleh Abdullahi, Eliot E. Miranda et Graem A. Ringwood — Collection schemes for distributed garbage. *In: Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 43–81 — septembre 1992, Springer-Verlag.

[4] Andrew W. Appel — Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, vol. 19, n° 2, février 1989, pp. 171–183.

[5] Andrew W. Appel — *Compiling with Continuations* — 1992, Cambridge University Press, England.

[6] Andrew W. Appel, John R. Ellis et Kai Li — Real-time concurrent collection on stock multiprocessors. *In: Programming Language Design and Implementation (PLDI)*. pp. 11–20 — juin 1988, ACM Press. (SIGPLAN Notices, vol. 23, n° 7). Version étendue: [39].

[7] Andrew W. Appel et Kai Li — *Virtual Memory Primitives for User Programs* — Rapport technique n° CS-TR-276-90, juillet 1990, Princeton University, New Jersey. Autre version de [8].

[8] Andrew W. Appel et Kai Li — Virtual memory primitives for user programs. *In: Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. pp. 96–107 — avril 1991, ACM Press. (SIGPLAN Notices, vol. 26, n° 4). Autre version: [7].

[9] S. Arnborg — Storage administration in a virtual memory SIMULA system. *BIT*, vol. 12, n° 2, 1972, pp. 125–141.

[10] J. L. Baer et H. Fries — On the efficiency of some list marking algorithms. *In: IFIP Congress*, éd. par Bruce Gilchrist. pp. 751–756 — août 1977, Elsevier Science Publishers B.V. (North-Holland).

[11] Henry G. Baker, Jr. — List processing in real time on a serial computer. *Communications of the ACM*, vol. 21, n° 4, avril 1978, pp. 280–294.

[12] Joel F. Bartlett — *Compacting Garbage Collection With Ambiguous Roots* — Rapport de recherche n° 88/2, février 1988, DEC Western Research Laboratory, Palo Alto, California. Version condensée: [13].

[13] Joel F. Bartlett — Compacting garbage collection with ambiguous roots. *LISP Pointers*, vol. 1, n° 6, avril 1988, pp. 3–12 — Version condensée de [12].

[14] Joel F. Bartlett — *Mostly-Copying Garbage Collection Picks Up Generations and C++* — Note n° TN-12, octobre 1989, DEC Western Research Laboratory, Palo Alto, California.

[15] Mordechai Ben-Ari — Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, vol. 6, n° 3, juillet 1984, pp. 333–344.

[16] D. I. Bevan — Distributed garbage collection using reference counting. *In: Parallel Architectures and Languages Europe (PARLE), vol. II: Parallel Languages*, éd. par J. W. de Bakker, A. J. Nijman et P. C. Treleaven. Lecture Notes in Computer Science (LNCS) n° 259, pp. 176–187 — juin 1987, Springer-Verlag.

[17] Andrew D. Birrell — *An Introduction to Programming with Threads* — Rapport de recherche n° 35, janvier 1989, DEC Systems Research Center, Palo Alto, California. Republié dans [18].

[18] Andrew D. Birrell — An introduction to programming with threads. *In: Systems Programming with Modula-3*, éd. par Greg Nelson, chap. 4, pp. 88–118 — Prentice-Hall, 1991. Republication de [17].

[19] Daniel G. Bobrow — Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, vol. 2, n° 3, juillet 1980, pp. 269–273.

[20] Hans-J. Boehm, Alan J. Demers et Scott Shenker — Mostly parallel garbage collection. *In: Programming Language Design and Implementation (PLDI)*. pp. 157–164 — juin 1991, ACM Press. (SIGPLAN Notices, vol. 26, n° 6).

[21] Rodney A. Brooks — Trading data space for reduced time and code space in real-time garbage collection on stock hardware. *In: LISP and Functional Programming (LFP)*. pp. 256–262 — août 1984, ACM Press.

[22] K. Mani Chandy et Jadayev Misra — *Parallel Program Design: A Foundation* — 1988, Addison-Wesley.

[23] C. J. Cheney — A nonrecursive list compacting algorithm. *Communications of the ACM*, vol. 13, n° 11, novembre 1970, pp. 677–678.

[24] Jacques Cohen — Garbage collection of linked data structures. *ACM Computing Surveys*, vol. 13, n° 3, septembre 1981, pp. 341–367.

[25] Eric C. Cooper et Richard P. Draves — *C Threads* — Rapport technique n° CMU-CS-88-154, juin 1988, Carnegie-Mellon University, Computer Science Department.

[26] Jeffrey L. Dawson — Improved effectiveness from a real time LISP garbage collector. *In : LISP and Functional Programming (LFP)*. pp. 159–167 — août 1982, ACM Press.

[27] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow et Scott Shenker — Combining generational and conservative garbage collection: Framework and implementations. *In : Principles of Programming Languages (POPL)*. pp. 261–269 — janvier 1990, ACM Press.

[28] David L. Detlefs — *Concurrent Garbage Collection for C++* — Rapport technique n° CMU-CS-90-119, mai 1990, CMU, Computer Science Department.

[29] John DeTreville — *Experience with Concurrent Garbage Collectors for Modula*-2^+ — Rapport de recherche n° 64, novembre 1990, DEC Systems Research Center, Palo Alto, California.

[30] Peter L. Deutsch et Daniel G. Bobrow — An efficient, incremental, automatic garbage collector. *Communications of the ACM*, vol. 19, n° 9, septembre 1976, pp. 522–526.

[31] E. W. Dijkstra — Co-operating sequential processes. *In : Programming Languages*, éd. par F. Genuys, pp. 43–112 — Academic Press, London, 1968.

[32] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten et E. F. M. Steffens — On-the-fly garbage collection: An exercice in cooperation. *In : Language Hierarchies and Interfaces*, éd. par Friedrich L. Bauer et Klaus Samelson, pp. 43–56 — Springer-Verlag, 1976. Première version de [33].

[33] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten et E. F. M. Steffens — On-the-fly garbage collection: An exercice in cooperation. *Communications of the ACM*, vol. 21, n° 11, novembre 1978, pp. 966–975 — Version étendue de [32].

[34] Amer Diwan, David Tarditi et Eliot Moss — Memory subsystem performance of programs using copying garbage collection. *In : Principles of Programming Languages (POPL)*. pp. 1–14 — janvier 1994, ACM Press.

[35] Damien Doligez — *Réalisation d'un Glaneur de Cellules de Lang et Dupont à Générations* — Rapport de DEA, septembre 1989, Université Paris 7.

[36] Damien Doligez et Georges Gonthier — Portable, unobtrusive garbage collection for multiprocessor systems. *In: Principles of Programming Languages (POPL)*. pp. 70–83 — janvier 1994, ACM Press.

[37] Damien Doligez et Xavier Leroy — A concurrent, generational garbage collector for a multithreaded implementation of ML. *In: Principles of Programming Languages (POPL)*. pp. 113–123 — janvier 1993, ACM Press.

[38] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Morhing, Benjamin Werner et al. — CoQ (logiciel) — URL= `ftp://ftp.inria.fr/INRIA/Projects/coq/coq/`, 1989. (192.93.2.54).

[39] John R. Ellis, Kai Li et Andrew W. Appel — *Real-time Concurrent Collection on Stock Multiprocessors* — Rapport de recherche n° 25, février 1988, DEC Systems Research Center, Palo Alto, California. Version étendue de [6].

[40] Robert R. Fenichel et Jerome C. Yochelson — A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, vol. 12, n° 11, novembre 1969, pp. 611–612.

[41] David A. Fisher — Copying cyclic list structure in linear time using bounded workspace. *Communications of the ACM*, vol. 18, n° 5, mai 1975, pp. 251–252.

[42] Benjamin Goldberg — Generation reference counting: A reduced-communication distributed storage reclamation scheme. *In: Programming Language Design and Implementation (PLDI)*. pp. 313–321 — juin 1989, ACM Press. (SIGPLAN Notices, vol. 24, n° 7).

[43] Georges Gonthier — Communication privée, août 1993.

[44] David Gries — An exercise in proving parallel programs correct. *In: Language Hierarchies and Interfaces*, éd. par Friedrich L. Bauer et Klaus Samelson, pp. 57–81 — Springer-Verlag, 1976. Première version de [45].

[45] David Gries — An exercise in proving parallel programs correct. *Communications of the ACM*, vol. 20, n° 12, décembre 1977, pp. 921–930 — Nouvelle version de [44].

[46] Robert H. Halstead, Jr. — Implementation of Multilisp: LISP on a multiprocessor. *In: LISP and Functional Programming (LFP)*. pp. 9–17 — août 1984, ACM Press. Version étendue: [47].

[47] Robert H. Halstead, Jr. — Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, n° 4, octobre 1985, pp. 501–538 — Version étendue de [46].

[48] Maurice Herlihy et J. Eliot B. Moss — *Non-Blocking Garbage Collection for Multiprocessors* — Rapport technique n° CRL 90/9, novembre 1990, DEC Cambridge Research Laboratory, Cambridge, Massachusetts.

[49] Yasushi Hibino — A practical parallel garbage collection algorithm and its implementation. *In : Symposium on Computer Architecture*. pp. 113–120 — mai 1980, ACM Press and IEEE Computer Society Press. (SIGARCH Newsletter / Computer Architecture News, vol. 8, n° 3).

[50] Paul Hudak — A semantic model of reference counting and its abstraction. *In : LISP and Functional Programming (LFP)*. pp. 351–363 — août 1986, ACM Press.

[51] John Hughes — A distributed garbage collection algorithm. *In : Functional Programming and Computer Architecture (FPCA)*, éd. par Jean-Pierre Jouannaud. Lecture Notes in Computer Science (LNCS) n° 201, pp. 256–272 — septembre 1985, Springer-Verlag.

[52] Neils-Christian Juul et Eric Jul — Comprehensive and robust garbage collection in a distributed system. *In : Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 103–115 — septembre 1992, Springer-Verlag.

[53] Jari Karjala — The world is just a huge fractal — Présenté à *International Obfuscated C Code Contest*, 1991.

```
/*-- Jari.Karjala@icl.fi -- The World is Just a Huge Fractal --*/
float o=0.075,h=1.5,T,r,O,l,I;int _,L=80,s=3200;main(){for(;s%L||
(h-=o,T= -2),s;4 -(r=O*O)<(l=I*I)|++ _==L&&write(1,(--s%L?_<L?--_
%6:6:7)+"World! \n",1)&&(O=I=l=_=r=0,T+=o /2))O=I*2*O+h,I=l+T-r;}
```

[54] H. T. Kung et S. W. Song — An efficient parallel garbage collection system and its correctness proof. *In : Foundations of Computer Science (FOCS)*. pp. 120–131 — octobre 1977, IEEE Computer Society Press.

[55] T. Kurokawa — *A New Fast and Safe Marking Algorithm* — Rapport technique, janvier 1979, Toshiba R&D Center, Kawasaki, Japon.

[56] Toshiaki Kurokawa — New marking algorithms for garbage collection. *In : USA-Japan Computer Conference*, pp. 580–584 — août 1975.

[57] Leslie Lamport — Garbage collection with multiple processes: An exercise in parallelism. *In : International Conference on Parallel Processing (ICPP)*, éd. par Philip H. Enslow Jr. pp. 50–54 — août 1976, IEEE Computer Society Press.

[58] Leslie Lamport — *A Temporal Logic of Actions* — Rapport de recherche n° 57, avril 1990, DEC Systems Research Center, Palo Alto, California. Première version de [59].

[59] Leslie Lamport — *The Temporal Logic of Actions* — Rapport de recherche n° 79, décembre 1991, DEC Systems Research Center, Palo Alto, California. Version étendue de [58] et [60].

[60] Leslie Lamport — The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 3, mai 1994, pp. 872–923 — Autre version de [59].

[61] Bernard Lang et Francis Dupont — Incremental incrementally compacting garbage collection. *In : Symposium on Interpreters and Interpretive Techniques.* ACM Press, pp. 253–263 — juin 1987. (SIGPLAN Notices, vol. 22, n° 7). Version révisée: [62].

[62] Bernard Lang et Francis Dupont — Incremental incrementally compacting garbage collection. *In : Programming of Future Generation Computers II*, éd. par Kazuhiro Fuchi et Laurent Kott. pp. 163–182 — 1988, Elsevier Science Publishers B.V. (North-Holland). Version révisée de [61].

[63] Bernard Lang, Christian Queinnec et José Piquer — Garbage collecting the world. *In : Principles of Programming Languages (POPL).* pp. 39–50 — janvier 1992, ACM Press.

[64] Thierry Le Sergent et Bernard Berthomieu — *Un ramasse miettes distribué incrémental sur une mémoire virtuelle partagée distribuée* — Rapport de recherche n° 91373, novembre 1991, LAAS-CNRS, Toulouse, France. Version française de [65].

[65] Thierry Le Sergent et Bernard Berthomieu — Incremental multi-threaded garbage collection on virtually shared memory architecture. *In : Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 179–199 — septembre 1992, Springer-Verlag. (Rapport de recherche LAAS-CNRS n° 92077) Version française: [64].

[66] Xavier Leroy — *The ZINC experiment: an economical implementation of the ML language* — Rapport technique n° 117, janvier 1990, INRIA, Rocquencourt, France.

[67] Xavier Leroy et al. — Caml Light (logiciel) — URL= `ftp://ftp.inria.fr/INRIA/Projects/cristal/caml-light/`, 1990. (192.93.2.54).

[68] Xavier Leroy et Pierre Weis — *Manuel de référence du langage Caml* — juillet 1993, InterÉditions, Paris, France.

[69] Henry Lieberman et Carl Hewitt — A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, vol. 26, n° 6, juin 1983, pp. 419–429.

[70] Gary Lindstrom — Copying list structures using bounded workspace. *Communications of the ACM*, vol. 17, n° 4, avril 1974, pp. 198–202.

[71] Barbara Liskov et Rivka Ladin — Highly-available distributed services and fault-tolerant distributed garbage collection. *In : Symposium on Principles of Distributed Computing.* pp. 29–39 — août 1986, ACM Press.

[72] J. Harold McBeth — On the reference counter method. *Communications of the ACM*, vol. 6, n° 9, septembre 1963, p. 575 — (lettre).

[73] John McCarthy — Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, vol. 3, n° 4, avril 1960, pp. 184–195.

[74] David A. Moon — Garbage collection in a large Lisp system. *In : LISP and Functional Programming (LFP).* pp. 235–246 — août 1984, ACM Press.

[75] J. Gregory Morriset et Andrew Tolmach — *A Portable Multiprocessor Interface for Standard ML of New Jersey* — Rapport technique n° CMU-CS-92-155, 1992, Carnegie-Mellon University, Computer Science Department.

[76] Scott Nettles, James O'Toole, David Pierce et Nicholas Haines — Replication-based incremental copying collection. *In : Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 357–364 — septembre 1992, Springer-Verlag.

[77] S. C. North et J. H. Reppy — Concurrent garbage collection on stock hardware. *In : Functional Programming and Computer Architecture (FPCA)*, éd. par Gilles Kahn. Lecture Notes in Computer Science (LNCS) n° 274, pp. 113–133 — septembre 1987, Springer-Verlag.

[78] G. L. Peterson — Myths about the mutual exclusion problem. *Information Processing Letters*, vol. 12, n° 3, juin 1981, pp. 115–116.

[79] Carl Pixley — An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, vol. 3, n° 1, décembre 1988, pp. 41–50.

[80] David Plainfossé et Marc Shapiro — Experience with fault tolerant garbage collection in a distributed Lisp system. *In : Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 166–133 — septembre 1992, Springer-Verlag.

[81] Christian Queinnec, Barbara Beaudoing et Jean-Pierre Queille — Mark DURING sweep rather than mark THEN sweep. *In : Parallel Architectures and Languages Europe (PARLE), vol. I: Parallel Architectures*, éd. par Eddy Odijik, Martin Rem et Jean-Claude Syre. Lecture Notes in Computer Science (LNCS) n° 365, pp. 224–237 — juin 1989, Springer-Verlag.

[82] John H. Reppy — CML: a higher-order concurrent language. *In: Programming Language Design and Implementation (PLDI)*. pp. 293–305 — juin 1991, ACM Press. (SIGPLAN Notices, vol. 26, n° 6).

[83] Martin Rudalics — Distributed copying garbage collection. *In: LISP and Functional Programming (LFP)*. pp. 364–372 — août 1986, ACM Press.

[84] Marc Shapiro — A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. *In: Symposium on Reliable Distributed Systems*. pp. 208–217 — septembre 1991, IEEE Computer Society Press.

[85] Marc Shapiro, Olivier Gruber et David Plainfossé — *A garbage detection protocol for a realistic distributed object-support system* — Rapport de recherche n° 1320, novembre 1990, INRIA, Rocquencourt, France.

[86] Ravi Sharma et Mary Lou Soffa — Parallel generational garbage collection. *In: Object-Oriented Programming Systems and Languages (OOPSLA)*, éd. par Andreas Paepcke. pp. 16–32 — octobre 1991, ACM Press. (SIGPLAN Notices, vol. 26, n° 11).

[87] Eric Spir — *Étude et Implantation d'un Glaneur de Cellules Adaptatif pour LISP* — Thèse de doctorat, janvier 1989, Université Paris 7.

[88] Guy L. Steele Jr. — Multiprocessing compactifying garbage collection. *Communications of the ACM*, vol. 18, n° 9, septembre 1975, pp. 495–508.

[89] Lars-Erik Thorelli — Marking algorithms. *BIT*, vol. 12, n° 4, 1972, pp. 555–568.

[90] David Ungar — Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *In: Software Engineering Symposium on Practical Software Development Environments*, éd. par Peter Henderson. pp. 157–167 — avril 1984, ACM Press. (SIGPLAN Notices, vol. 19, n° 5; Software Engineering Notes, vol. 9, n° 3).

[91] David Ungar et Frank Jackson — Tenuring policies for generation-based storage reclamation. *In: Object-Oriented Programming Systems and Languages (OOPSLA)*, éd. par Norman Meyrowitz. pp. 1–17 — septembre 1988, ACM Press. (SIGPLAN Notices, vol. 23, n° 11).

[92] Stephen C. Vestal — *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming* — Thèse de PhD, janvier 1987, University of Washington, Seattle, Washington. (Rapport Technique n° 87-01-03).

[93] Philip L. Wadler — Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, vol. 19, n° 9, septembre 1976, pp. 491–500.

[94] David C. Walden — A note on Cheney's nonrecursive list-compacting algorithm. *Communications of the ACM*, vol. 15, n° 4, avril 1972, p. 275.

[95] Pierre Weis et Xavier Leroy — *Le langage Caml* — juillet 1993, InterÉditions, Paris, France.

[96] E. P. Wentworth — Pitfalls of conservative garbage collection. *Software—Practice and Experience*, vol. 20, n° 7, juillet 1990, pp. 719–727.

[97] Paul R. Wilson — A simple bucket-brigade advancement mechanism for generation-based garbage collection. *SIGPLAN Notices*, vol. 24, n° 5, mai 1989, pp. 38–46.

[98] Paul R. Wilson — Uniprocessor garbage collection techniques. *In: Memory Management: International Workshop (IWMM)*, éd. par Yves Bekkers et Jacques Cohen. Lecture Notes in Computer Science (LNCS) n° 637, pp. 1–42 — septembre 1992, Springer-Verlag.

[99] Paul R. Wilson, Michael S. Lam et Thomas G. Moher — Caching considerations for generational garbage collection. *In: LISP and Functional Programming (LFP)*. pp. 32–42 — juin 1992, ACM Press.

[100] David S. Wise — Morris's garbage compaction algorithm restores reference counts. *ACM Transactions on Programming Languages and Systems*, vol. 1, n° 1, juillet 1979, pp. 115–120.

[101] David S. Wise et Daniel P. Friedman — The one-bit reference count. *BIT*, vol. 17, n° 3, 1977, pp. 351–359.

[102] Taiichi Yuasa — *Realtime Garbage Collection on General-purpose Machines* — Rapport de recherche n° 535, février 1986, Research Institute for Mathematical Sciences, Kyoto University, Japon. Nouvelle version: [103].

[103] Taiichi Yuasa — Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, vol. 11, n° 3, mars 1990, pp. 181–198 — Nouvelle version de [102].

[104] Benjamin Zorn — Comparing mark-and-sweep and stop-and-copy garbage collection. *In: LISP and Functional Programming (LFP)*. pp. 87–98 — juin 1990, ACM Press.

Index

Résumé

La gestion manuelle de la mémoire est une source inépuisable d'erreurs faciles à commettre et difficiles à repérer. La gestion automatique de la mémoire libère le programmeur de ces erreurs, et un langage de haut niveau ne se conçoit plus sans gestion automatique de la mémoire. Dans le cas d'une machine parallèle, la gestion de la mémoire devient un problème délicat. Cette difficulté augmente encore l'intérêt d'un gestionnaire de mémoire automatique (GC).

Cette thèse est consacrée à la conception, la preuve et l'implémentation d'un GC destiné à être utilisé avec des programmes écrits dans un langage de haut niveau autorisant l'exécution parallèle avec un modèle à mémoire partagée. C'est un GC concurrent qui utilise au minimum les primitives de synchronisation coûteuses. Comme pour tout algorithme parallèle, il est difficile d'avoir confiance dans cet algorithme si on ne dispose pas d'une preuve formelle de sa correction. Nous donnons une telle preuve, en utilisant le formalisme TLA.

Nous donnons aussi une extension de cet algorithme par un système de générations, qui permet d'obtenir des performances très satisfaisantes. Cette extension utilise de façon originale le typage statique fort du langage ML pour faire coopérer le programme avec le GC. Cette extension pourrait aussi s'adapter à d'autres GC concurrents ou parallèles.

Mots-clés

Gestion de mémoire. GC. Glaneur de cellules. ML. GC à générations. GC concurrent. Parallélisme. Mémoire partagée. Preuve de programme. TLA.