Vérification formelle d'une implémentation d'un gestionnaire de mémoire pour un compilateur certifié

Tahina Ramananandro
sous la direction de Xavier Leroy

Stage de Master 2
Master Parisien de Recherche en Informatique
19 février – 27 juillet 2007

Table des matières

1 Introduction

1.1 Le compilateur CompCert

Depuis 2005, l'INRIA Rocquencourt abrite en son sein une équipe projet, nommée GALLIUM et dirigée par Xavier LEROY, qui s'intéresse à la *compilation certifiée* : prouver que, étant donnés les sémantiques des langages source et cible d'un compilateur, et sous l'hypothèse que le compilateur termine correctement, alors la valeur de retour et la trace d'exécution (tous les *événements* d'exécution tels que les entrées-sorties) de tout programme sont les mêmes pour le code source et pour le code produit par le compilateur.

Il existe plusieurs approches de compilation certifiée. Parmi celles-ci, GALLIUM s'intéresse au développement d'un *compilateur certifié* : le processus de compilation lui-même est prouvé correct. Le but n'est pas seulement de montrer qu'on peut compiler un programme de façon sûre vis-à-vis des sémantiques, mais aussi de montrer qu'on peut écrire un compilateur *optimisant* certifié correct – sans toutefois prétendre prouver toutes les optimisations des compilateurs courants tels que `gcc`.

Notons que comme la compilation certifiée est un problème indécidable en général, on cherche uniquement à avoir des réponses partielles : le compilateur peut ne pas terminer, ou terminer en erreur, et dans ces deux cas on peut avoir des fausses alertes non détectées par la certification.

L'équipe GALLIUM développe actuellement un compilateur certifié, nommé *CompCert* [Ler06][ea07], du langage C vers le langage machine de l'architecture PowerPC, à l'aide de l'assistant de preuve COQ [Coq].

Le compilateur CompCert procède par traductions successives à travers des langages intermédiaires. Il se divise en deux parties, *front-end* et *back-end*, autour du langage intermédiaire *Cminor*, un sous-ensemble de C.

On appelle *back-end* le compilateur de Cminor vers le langage machine PowerPC. C'est la composante majeure du compilateur CompCert.

En revanche, il n'y a pas lieu de parler d'un unique *front-end*. Il y a plutôt un front-end pour chaque langage source qu'on veut compiler. Un front-end compile donc vers le langage Cminor. Il y a donc un front-end pour le langage C [BDL06], mais aussi un front-end pour MiniML, qui est un sous-ensemble d'OCaml.

Or, le compilateur CompCert est développé au moyen de la fonction d'*extraction* de programmes à partir des preuves de correction écrites en COQ : la correction du compilateur est constituée de théorèmes dont l'extraction des preuves produit le compilateur lui-même. Le noyau de l'assistant de preuve COQ (en gros, un *type-checker* qui vérifie que les preuves sont bien formées et correspondent bien aux théorèmes à prouver) a déjà été prouvé correct dans COQ lui-même [Bar99], mais la procédure d'extraction doit être certifiée elle aussi. C'est bien le cas si l'on choisit l'extraction vers le langage MiniML [Let04]. Il ne reste donc plus qu'à compiler les programmes MiniML ainsi extraits. Cette étape de compilation devant aussi être certifiée, cela justifie le fait de vouloir certifier un front-end pour ce langage.

1.2 Gestion de la mémoire pour les langages de haut niveau

1.2.1 Gestionnaire de mémoire

MiniML est un langage de haut niveau : à l'instar d'OCaml, ou même de Java, et contrairement à C, les allocations et libérations mémoire ne sont pas explicites : dans un tel langage de haut

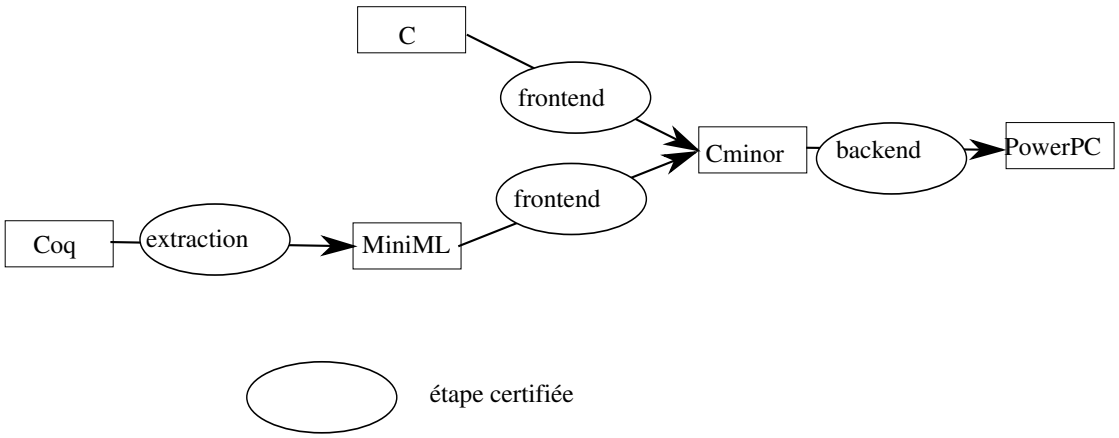

FIG. 1 – Le compilateur CompCert

niveau, le programmeur n'a pas à se soucier de l'allocation de mémoire lorsqu'il veut créer un objet, une fermeture fonctionnelle, etc., et à l'inverse, de sa libération lorsqu'il veut les détruire. C'est l'environnement d'exécution du langage de haut niveau qui s'en charge, à travers un gestionnaire de mémoire.

L'environnement d'exécution constitue la majeure partie de l'interpréteur (dans le cas de langages interprétés tels que BASIC) ou de la machine virtuelle (dans le cas de langages à *bytecode* tels que OCaml ou Java). Cependant, si l'on souhaite compiler « en code natif », c'est-à-dire produire du vrai code machine, par exemple via le back-end CompCert, l'environnement d'exécution doit pouvoir être intégré à ce compilateur. Or :

- dans la chaîne de compilation de MiniML vers le langage machine PowerPC, Cminor est le premier langage de programmation qui fasse intervenir de la gestion mémoire explicite (c'est-à-dire, où apparaissent explicitement des appels à des fonctions telles que `malloc` et `free`). Donc, il n'est pas possible d'écrire le gestionnaire de mémoire dans un langage de plus haut niveau que Cminor.
- à l'inverse, il serait impossible d'écrire le gestionnaire de mémoire dans un langage de plus bas niveau tel que l'assembleur PowerPC, à l'instar de [MSLL07], à cause des optimisations par le back-end du code Cminor produit par le front-end, optimisations qu'il faudrait alors prendre en compte dans la preuve de certification du gestionnaire de mémoire. Cette preuve pourrait alors être rendue inutile dans l'éventuel cas où l'on développerait un autre back-end, par exemple pour une autre architecture. Aussi le gestionnaire de mémoire doit-il être intégré autant que possible au front-end.

C'est pourquoi le gestionnaire de mémoire est écrit directement en Cminor, et est ajouté au code Cminor produit par le front-end, le tout passant ensuite sous traitement par le back-end.

Il faut alors prouver que ce code Cminor rajouté ne modifie pas la sémantique du programme source original écrit en langage de haut niveau.

Le gestionnaire de mémoire est *a priori* composé de deux parties :

1. Lorsque le programme veut créer un objet, l'*allocateur* attribue l'espace mémoire requis pour cet objet.
2. Une fois cet espace mémoire alloué, il reste alors à construire l'objet proprement dit, selon le code source en langage de haut niveau. C'est le rôle du *mutateur*.

Il faut donc prouver que l'allocateur :

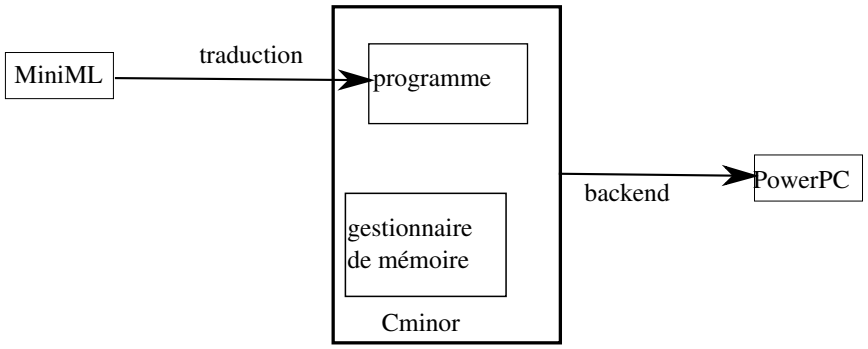

FIG. 2 – Compilation d'un programme MiniML vers PowerPC via Cminor

– conserve la structure de tas de la mémoire
– alloue un objet bien formé du nouveau tas, qui n'est pas parmi les objets accessibles de l'ancien tas, et qui n'est pas dans la freelist du nouveau tas
– conserve l'invariant que les objets de la freelist ne sont pas accessibles

Il faut aussi prouver que le mutateur affecte bien comme objets fils du nouvel objet des objets accessibles dans la mémoire avant mutation.

Exemple. Listes d'entiers.

En OCaml, une liste d'entiers est une construction inductive à deux constructeurs : nil, une constante correspondant à la liste vide, et cons, qui enchaîne un entier à une liste déjà existante. En mémoire, on peut représenter une liste comme suit :

– nil, une "case mémoire" à un endroit fixé de la mémoire
– cons, comme un objet à deux "cases mémoire", l'une contenant l'entier à enchaîner, l'autre contenant un pointeur vers la suite de la liste : vers *nil* ou vers la première "case mémoire" d'un objet cons.

Lorsque le programmeur veut créer, par exemple, la liste 18 :: 42 :: nil, voici comment procède le gestionnaire de mémoire :

1. avant même l'exécution du programme, un endroit fixe est décidé pour la case mémoire correspondant à nil
2. l'allocateur mémoire demande un espace de deux cases mémoire pour la construction de 42 :: nil
3. le mutateur initialise les cases mémoire ainsi allouées, en stockant dans la première l'entier 42,
4. et un pointeur vers la case nil dans la seconde.
5. l'allocateur mémoire demande ensuite un espace de deux cases mémoire pour la construction de 18 :: (42 :: nil), où 42 :: nil a déjà été construit.
6. Le mutateur initialise ensuite les cases mémoire ainsi allouées en stockant dans la première l'entier 18,
7. et, dans la seconde, un pointeur vers, par exemple, la première des deux cases mémoire correspondant à l'objet 42 :: nil.

1.2.2 Ramasse-miettes (*garbage collector* ou GC)

Principe Si un gestionnaire de mémoire ne permettait que de créer des objets, qui ne seraient détruits qu'à la fin de l'exécution du programme, il faudrait alors que le programme à exécuter

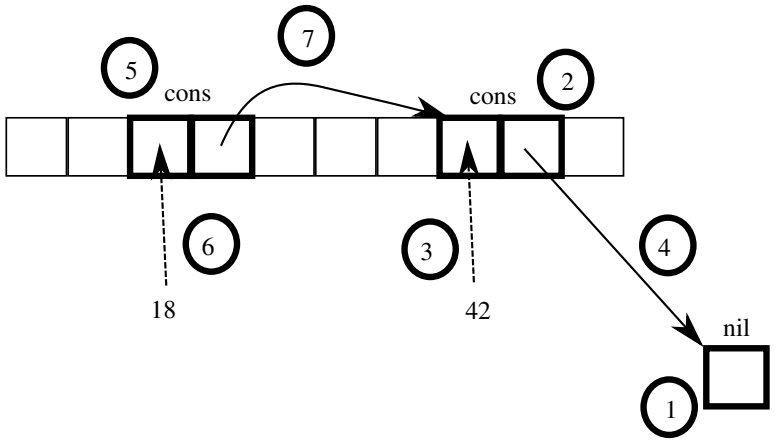

FIG. 3 – Gestion de mémoire : création d'une liste

dispose, dès le départ, d'une quantité de mémoire au moins égale au total des tailles de *tous* les objets manipulés, même s'ils ne sont pas manipulés en même temps. Or, cela serait impensable si le programme devait par exemple manipuler des fichiers de plusieurs giga-octets dans le cas où il devrait lire le fichier octet par octet (par exemple, recherche d'une chaîne). Il faudrait donc, pour chaque octet lu, que le programme crée l'objet lui correspondant, donc alloue la mémoire correspondante. Or, dans la plupart des cas, le programme n'a plus besoin que des derniers octets lus.

Cela n'est donc pas suffisant. Il faut donc adjoindre à l'allocateur un procédé de récupération de la mémoire occupée par des objets créés auparavant mais qui ne sont plus utilisés par le programme. C'est ce qu'on appelle un *garbage collector* (GC) : « récupérateur de déchets », en français *ramasse-miettes*[1].

Cependant, pour pouvoir écrire un ramasse-miettes, il faut d'abord définir ce qu'est un objet utilisé. Or, même au niveau du code source du programme, savoir si une variable est utilisée est indécidable en général. Donc, a fortiori, savoir si un objet est utilisé est indécidable. On choisit donc une définition faible d'un objet utilisé.

On dit qu'un objet b est *fils* d'un objet a si, et seulement si, dans a, il existe un pointeur vers b. Alors, on dit qu'un objet est *utilisé* s'il est descendant, par cette filiation, d'un objet *racine*. Reste à définir la notion d'objet racine.

Dans [MSLL07], les objets racine sont uniquement ceux tels qu'il existe un pointeur vers eux dans les registres. Cette définition orientée vers le langage cible compilé est fort adaptée à des gestionnaires de mémoire écrits dans un langage assembleur. Cependant, ce n'est pas notre cas ici. Il faut plutôt choisir une définition orientée « dans l'autre sens », c'est-à-dire vers le langage à compiler. Dans un langage tel que Java, par exemple, les objets racine sont les objets visibles dans le *scope* au moment de l'exécution : les objets assignés aux variables locales de chaque procédure de la pile d'exécution, ainsi qu'aux variables globales au programme.

Une fois que les objets utilisés sont déterminés, il s'agit de récupérer l'espace laissé libre afin de le mettre à la disposition de l'allocateur. À cette fin, les ramasse-miettes créent en général une *freelist*, liste des objets libres, que l'allocateur proprement dit n'a plus qu'à parcourir ; pour chaque objet rencontré durant ce parcours, il compare alors sa taille avec la taille requise, et renvoie le premier qui convienne (*first-fit*), ou bien peut maintenir la liste triée dans le sens des tailles croissantes, ou décroissantes (*best-fit*). Cependant, cette dernière approche est algorithmiquement plus lente. Le parcours d'une freelist doit, en effet, être plus rapide que le

[1] On trouve aussi la dénomination "glaneur de cellules", pour coller aux initiales...

parcours du tas entier. La freelist peut être stockée soit sous la forme d'une liste de pointeurs vers chaque objet libre, soit directement dans le tas, sous la forme d'une liste chaînée où chaque objet libre de la liste contient en guise de données un pointeur vers l'objet suivant dans la liste.

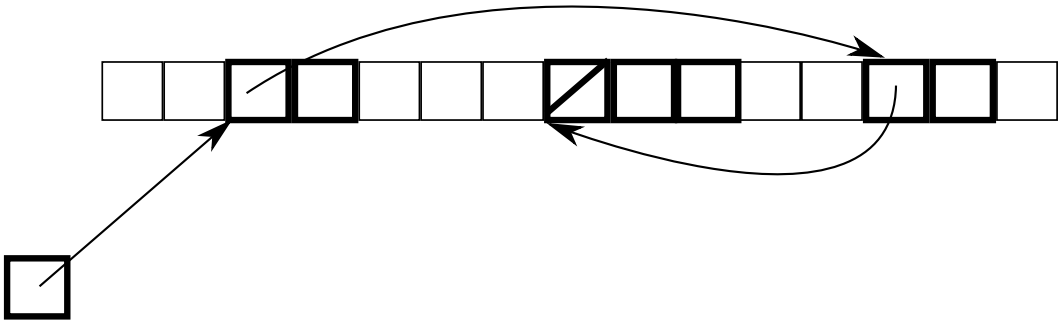

FIG. 4 – Liste chaînée d'objets du tas. La tête de la liste est stockée hors du tas.

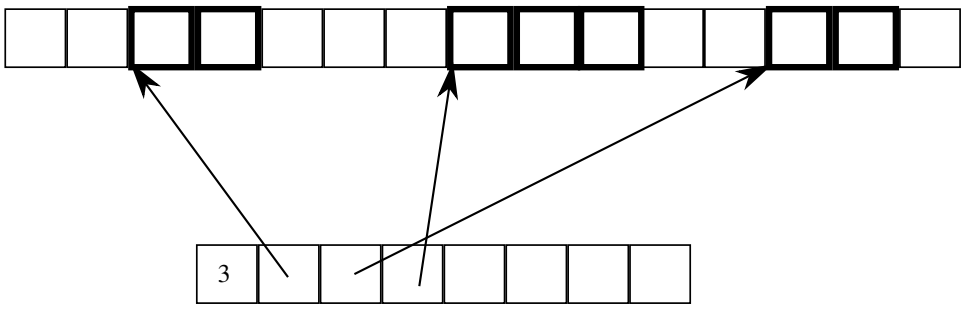

FIG. 5 – Liste d'objets du tas : liste de pointeurs stockée hors du tas.

Gérer une liste à part a l'avantage de ne pas modifier les données du tas, mais nécessite cependant plus d'espace mémoire ; de plus, il faut connaître à l'avance une borne sur la taille de cette liste, au risque de ne stocker qu'une liste partielle d'objets libres. Dans le cas de la gestion de la freelist directement dans le tas, il faut exclure les objets de taille nulle (ils ne peuvent admettre aucune donnée supplémentaire que l'en-tête, donc on ne peut pas y stocker de pointeur vers l'objet suivant).

Les grandes familles de ramasse-miettes Il existe de très nombreux algorithmes de ramasse-miettes. Les plus connus sont décrits dans [Wil92]. Cependant, on peut en dégager deux grandes familles, les *mark and sweep* et les *stop and copy*.

Les ramasse-miettes *mark and sweep* (« marquer et nettoyer ») procèdent en *marquant* les objets utilisés, puis en nettoyant le tas en le balayant pour ajouter à la freelist chaque objet non marqué, et effacer la marque des objets utilisés. Le marquage peut être récursif : dans ce cas, il utilise alors deux couleurs, le *blanc* (aucune marque) pour les objets non utilisés, et le *noir* pour les objets utilisés. Pour chaque objet utilisé non encore marqué (à commencer par les racines), le GC le marque en noir, puis marque récursivement chacun de ses fils (parcours en profondeur d'abord). Mais ce marquage récursif nécessite alors une pile d'exécution de taille importante. C'est pourquoi on peut aussi envisager un marquage à *cache* : le GC met en cache chaque objet racine sans le marquer, puis, pour chaque objet du cache non encore marqué, le GC le sort du cache, le marque et met en cache tous ses fils qui n'y sont pas encore, jusqu'à épuiser le cache. Le cache peut être implicite, sous la forme d'une troisième couleur, le *gris*, pour les objets mis en cache, c'est-à-dire déjà rencontrés mais en attente d'être marqués. Mais

alors, pour chercher un élément du cache, il faut parcourir tout le tas en quête d'un objet gris. Cela peut prendre du temps. On peut donc opter pour un cache explicite, sous la forme d'une liste de pointeurs à l'extérieur du tas, manipulée comme une file ou comme une pile. Cependant, il faut connaître une borne sur la taille de cette liste. De plus, savoir si un objet est déjà mis en cache prend ici un temps linéaire (car il faut *a priori* parcourir toute la liste), alors qu'il est constant dans le cas du cache implicite (simple contrôle de la couleur). On peut alors combiner ces deux méthodes de mise en cache.

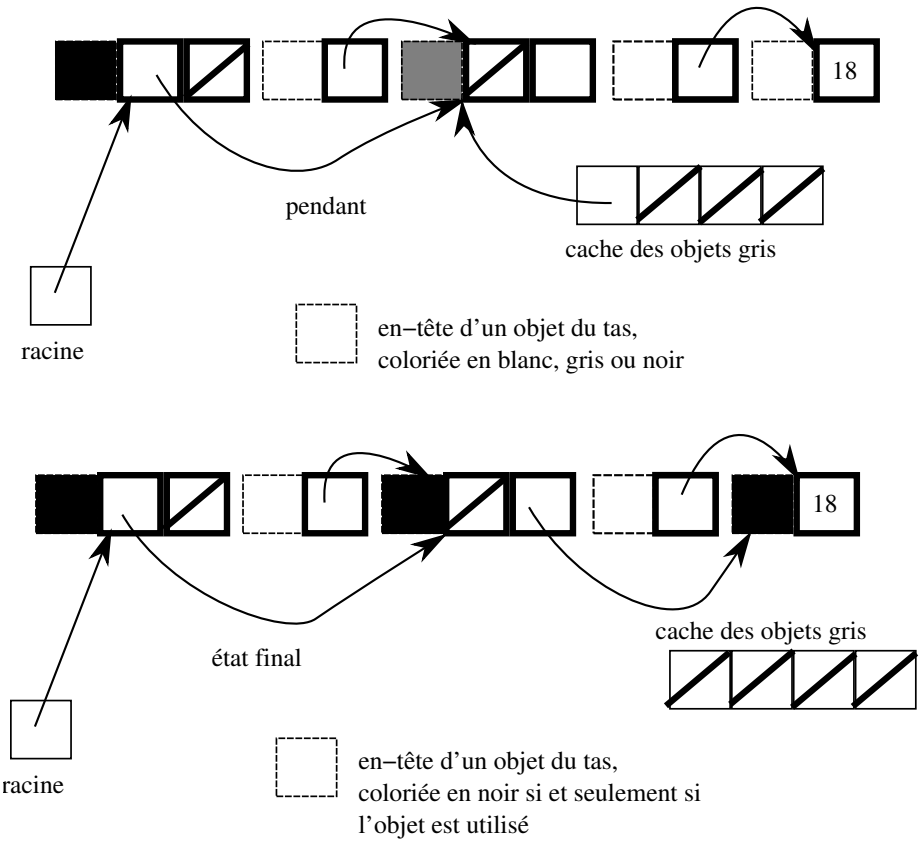

FIG. 6 – Mark and sweep mixte avec cache et marquage à trois couleurs

Le mark and sweep présente l'avantage non négligeable, du point de vue de la preuve, de conserver intacte la structure du tas : la taille, la disposition, mais aussi les données (sauf pour les objets libres) sont conservées. Cependant, l'étape de sweep peut également *coalescer* les objets libres, c'est-à-dire fusionner les objets libres adjacents du tas. Ceci permet d'allouer des espaces plus importants qu'auparavant, mais complique la preuve en modifiant la structure du tas.

Les ramasse-miettes *stop and copy* (« s'arrêter et copier ») procèdent en créant un nouveau tas vers lequel *copier* les objets utilisés. La copie peut être récursive : pour chaque objet utilisé non encore copié (à commencer par les racines), on le copie dans le nouveau tas, puis on copie récursivement chacun de ses fils. Ceci nécessite de marquer, dans le tas de départ, les objets déjà copiés. Mais lors de la copie d'un objet contenant des pointeurs, il faut également prendre garde à copier les pointeurs des nouveaux objets. La copie récursive peut poser quelques problèmes de ce point de vue. Donc, on peut opter pour une copie séquentielle, où deux pointeurs évoluent dans le tas cible : on copie d'abord *littéralement* (sans changer les pointeurs) en faisant avancer un pointeur de copie, et lorsque l'objet est copié, on met à jour ses pointeurs en faisant avancer un pointeur de mise à jour, sachant que pour chaque pointeur lu, on copie éventuellement l'objet référencé mais sans mettre à jour ses pointeurs. Le pointeur de copie et le pointeur de mise à

jour sont globaux, ils ne sont pas propres à chaque objet copié.

Le stop and copy modifie profondément la structure du tas. Il faut alors raisonner localement, en considérant des chemins qui traversent les deux tas en même temps. Cependant, c'est la solution optimale du point de vue de l'espace libre qu'on peut allouer, car à l'issue de ce GC, tout l'espace libre est contigu et forme un seul objet non utilisé. Toutefois, elle requiert la présence de deux fois l'espace requis pour le tas, puisqu'il faut copier les objets dans un nouveau tas.

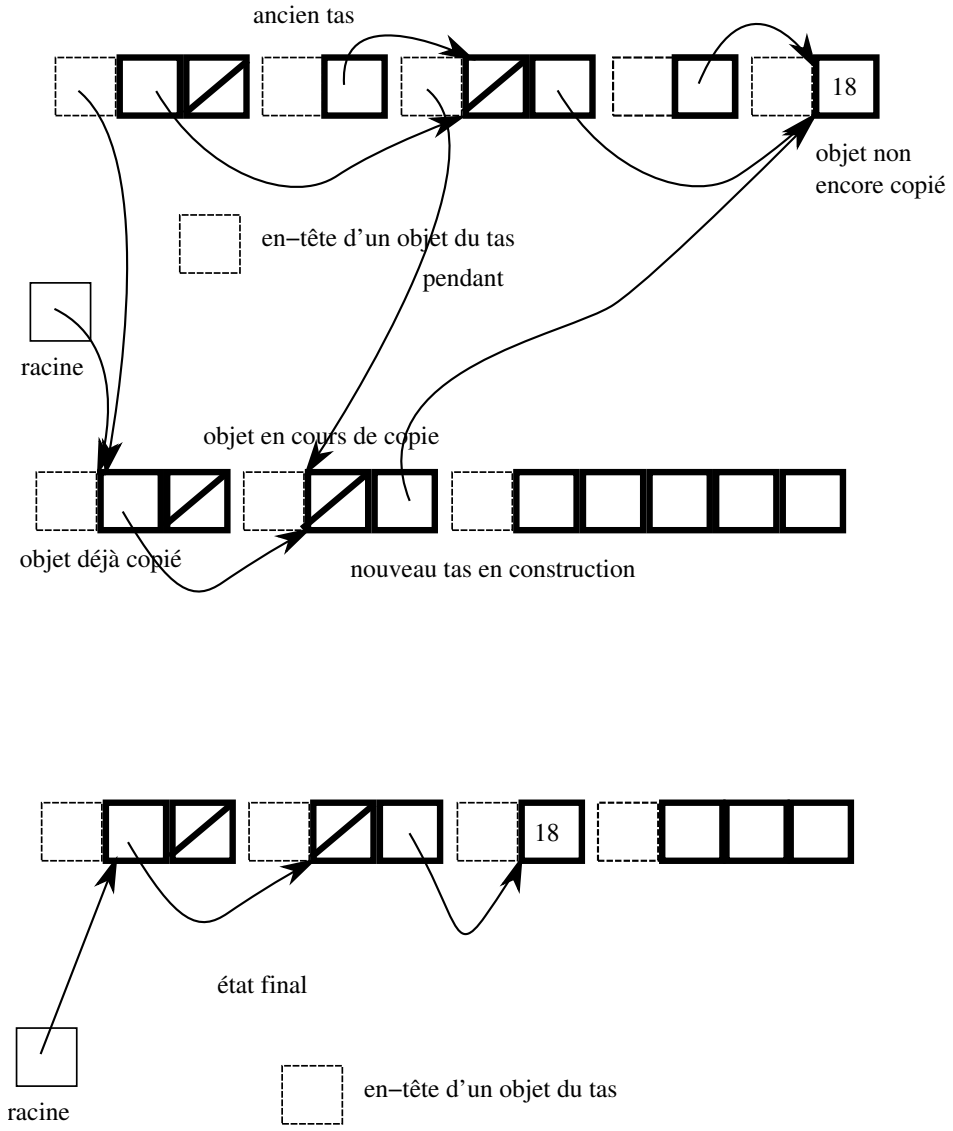

FIG. 7 – Stop and copy

On pourrait alors penser que pour chaque allocation, on pourrait lancer un GC stop and copy, pour être sûr de pouvoir allouer la taille demandée dès lors qu'elle n'est pas plus grande que le total de l'espace libre. Cependant, algorithmiquement, un tel GC est extrêmement lent.

A priori, les GC, tant mark and sweep que stop and copy, suspendent l'exécution du programme pendant leurs opérations : ce sont des GC *stop-the-world* (« arrêter le monde »). On peut éventuellement les rendre incrémentaux, de sorte qu'on puisse les interrompre, ou encore les rendre parallèles en les faisant s'exécuter dans une autre *thread*, mais il faut alors prendre garde à ne pas perturber la structure du tas et de la freelist, notamment dans le cas où une allocation survient alors que le GC n'a pas terminé sa tâche.

1.3 Le modèle mémoire

Le modèle mémoire présenté ici, sur lequel ce stage est fondé, a été formalisé par Xavier Leroy et Sandrine Blazy dans [BL05]. Ce modèle mémoire est utilisé par la plupart des langages intermédiaires de CompCert, en particulier Cminor. Il correspond au module Mem du développement Coq de CompCert.

1.3.1 Présentation

Une mémoire est un ensemble de « cases mémoire » d'un octet chacune. Ces cases mémoire sont regroupées en *blocs* indicés par des entiers relatifs, et au sein de chaque bloc, les cases mémoire sont repérées par un *décalage* ou *offset* par rapport à une case de référence. Cet offset est également un entier relatif.

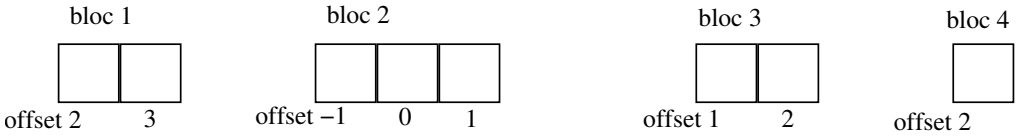

Fig. 8 – Mémoire organisée en blocs et offsets

Un bloc peut être alloué statiquement lors de la détermination des variables globales du programme en Cminor. Mais il peut aussi être créé dynamiquement, par un appel à une fonction d'allocation bas niveau telle que `malloc`. Il peut être libéré par une fonction de libération telle que `free`. Ceci permet de modéliser de façon abstraite les appels à `malloc` et `free` : le programme n'a pas besoin de connaître les adresses absolues.

Une mémoire contient un nombre *fini* de blocs. Cependant, les programmes peuvent allouer un nombre arbitraire de blocs : une telle allocation n'échoue jamais. Chaque bloc contient également un nombre *fini* de cases mémoire, nombre qui toutefois est donné au moment de son allocation. Au sein d'un bloc, les cases mémoire sont contiguës : l'ensemble des offsets valides d'un bloc mémoire est un intervalle de $\mathbb{Z}$. La taille du bloc ne peut plus être changée, sauf éventuellement au moment de sa libération : dans ce cas, la taille est mise à zéro, et plus aucune case mémoire du bloc n'est valide.

Le stockage de valeurs en mémoire s'effectue à un niveau abstrait. En effet, bien que chaque case représente un octet, la valeur de cet octet n'apparaît pas en tant que telle dans le modèle mémoire, mais c'est une valeur abstraite qui est représentée. Il ne suffit pas de stocker la valeur dans la première case mémoire, car écrire une valeur peut écraser une valeur stockée auparavant un ou deux octets plus loin. Pour tenir compte de tels effets, on stocke, dans les $n-1$ cases suivantes, un *tag* spécial, appelé cont, pour « data *cont*inued », c'est-à-dire que les données correspondant à la valeur lue plus haut continuent sur cette case mémoire. Pour lire une valeur de n octets à un offset o d'un bloc b, il faudra donc non seulement lire la valeur présente à la case o du bloc b, mais aussi contrôler que les $n-1$ cases $o+1, \ldots, o+n-1$ du bloc b contiennent bien des tags cont. Si ce n'est pas le cas, c'est-à-dire si une valeur a été lue parmi ces $n-1$ cases mémoire, cela signifie que la valeur lue dans la case o a été écrasée, partiellement ou totalement. À la lecture comme à l'écriture, il faut bien sûr contrôler que les cases mémoire concernées soient toutes valides au sein du bloc.

Il y a quatre sortes de valeurs :

- Vint, entier de 32 bits
- Vfloat, flottant double précision
- Vptr, pointeur. Un pointeur est constitué d'un numéro de bloc et d'un entier de 32 bits représentant l'offset. Ainsi, bien que les cases mémoire soient indexés par un offset entier relatif, au plus 2^{32} sont accessibles par pointeur, même si l'intervalle de validité du bloc est plus grand.
- Vundef, valeur non définie

Pour lire/écrire une valeur, il faut spécifier un *chunk*, qui indique non seulement la taille des données mais aussi le traitement de leur signe. En effet, les programmes peuvent stocker des entiers de 1, 2 ou 4 octets, signés ou non. Cependant, comme il n'existe en fait qu'un seul type d'entier, des *casts* sont nécessaires à chaque écriture. Il en est de même pour les flottants : les programmes peuvent stocker des flottants simple précision, la procédure d'écriture s'occupe du cast. (Cf. figure 9). Si le chunk et la valeur ne correspondent pas, ou si dans la mémoire il y a moins de tags cont que ne le requiert la lecture (mais si les cases mémoire lues sont quand même valides), alors la lecture en mémoire renvoie la valeur non définie Vundef.

Valeur	Arguments	Chunk	Octets	Description
Vint	Entier 32 bits	Mint8signed	1	Entier de 8 bits signé
		Mint8unsigned	1	Entier de 8 bits non signé
		Mint16signed	2	Entier de 16 bits signé
		Mint16unsigned	2	Entier de 16 bits non signé
		Mint32	4	Entier de 32 bits
Vfloat	flottant	Mfloat32	4	Flottant simple précision
		Mfloat64	8	Flottant double précision
Vptr	numéro de bloc + entier 32 bits	Mint32	4	Pointeur (bloc + offset)

FIG. 9 – Valeurs et chunks mémoire corrects

Le fait que l'allocation de bloc n'échoue jamais, ainsi que l'utilisation d'un chunk Mint32 pour les pointeurs, peuvent paraître absurdes. Cependant, il s'agit là d'un modèle mémoire assez abstrait, et le passage à une allocation réelle et à des pointeurs absolus (de « vrais » entiers 32 bits comme en C) s'effectue dans le back-end : dans ce cas, la mémoire est en gros organisée en un seul bloc.

Dans le cas de notre ramasse-miettes, seuls des chunks Mint32 (entiers 32 bits et pointeurs) sont utilisés.

1.3.2 Formalisation en Coq

Structure Le module Mem fournit les types block pour modéliser le numéro d'un bloc (en fait, c'est simplement Z, le type des entiers relatifs) et mem pour modéliser une mémoire. Une mémoire est la donnée :

- d'une fonction block_contents, qui associe à chaque numéro de bloc un bloc
- du numéro nextblock, entier relatif mais positif, du prochain bloc à allouer

Un bloc est la donnée de son contenu et de deux entiers relatifs lo et hi correspondant aux bornes du bloc, entre lesquelles les cases mémoire peuvent être lues et écrites.

Definition block : **Set** :=Z.
Record block_contents : **Set** :=mkblock **{** low : Z ; high : Z ; contents : contentmap **}**.
Record mem : **Set** := mkmem **{** blocks : block → block_contents ; nextblock : block ; nextblock_pos : nextblock > 0 **}**.
Le type contentmap décrivant le contenu des blocs sera décrit plus loin (figure 11 page 14)

FIG. 10 – Formalisation des blocs et mémoires

Alors, au sein d'une mémoire, un bloc est *valide* si et seulement si son numéro est strictement inférieur à nextblock. Au sein d'un bloc, un offset est *valide* si et seulement si il est compris entre low inclus et high exclu.

Le module Mem fournit deux fonctions : alloc pour l'allocation d'un bloc mémoire et free pour la libération d'un bloc mémoire.

Un appel à alloc s'écrit : alloc *mem lo hi*, où *mem* est une mémoire, et *lo* et *hi* des entiers relatifs représentant les offsets respectivement minimum et maximum du bloc à allouer. Le type de retour est mem×block : la nouvelle mémoire, et le numéro du nouveau bloc alloué (en fait, nextblock *mem*). Ainsi, l'allocation n'échoue jamais. Alors, le champ nextblock de la mémoire est incrémenté, et les champs low et high du nouveau bloc mémoire alloué sont initialisés respectivement à *lo* et *hi* ; ce nouveau bloc mémoire est la valeur de blocks *mem* (nextblock *mem*), les autres valeurs de blocks étant inchangées.

Un appel à free s'écrit : free *mem block*, où *mem* est une mémoire, et *block* un entier relatif représentant le numéro du bloc à libérer. Une libération fonctionne toujours, même s'il s'agit d'un bloc non valide. Elle retourne alors la nouvelle mémoire *mem'* telle que blocks *mem' block* est un bloc dont les champs low et high sont mis à zéro, les autres valeurs de blocks étant inchangées.

Le modèle mémoire prévoit alors les théorèmes régissant l'allocation et la libération d'un bloc mémoire :

- lors d'une allocation, les blocs valides dans la nouvelle mémoire sont le nouveau bloc alloué et les blocs valides de la mémoire avant allocation
- lors d'une allocation, les offsets valides des blocs valides de l'ancienne mémoire sont toujours valides
- lors d'une libération, les blocs valides restent les mêmes
- lors d'une libération, les offsets valides des blocs autres que le bloc libéré sont les mêmes

Stockage de valeurs Le module Mem fournit deux fonctions : load pour la lecture et store pour l'écriture.

Un appel à load s'écrit : load *chunk mem block offset*, où *chunk* est un chunk, *mem* une mémoire, et *block* et *offset* des entiers relatifs représentant respectivement le bloc et l'offset où lire la valeur. Le type de retour est option val en fonction de la validité du bloc et des offsets à lire :

- Si le bloc n'est pas valide (i.e n'a pas encore été alloué), ou si le bloc est valide mais que l'offset (ou celui de l'une des $n-1$ cases mémoire suivantes) n'est pas valide (i.e. est en dehors de l'intervalle de validité du bloc), alors l'appel renvoie None.
- Si le bloc est valide et l'offset au sein du bloc est aussi valide, et que les $n-1$ cases mémoire suivantes sont valides mais n'ont pas toutes le tag cont (c'est-à-dire que l'une d'entre elles contient une valeur), alors l'appel renvoie Some Vundef. De même si la case mémoire demandée a le tag cont (dans ce cas, il n'y a pas de valeur à lire).
- Si le bloc est valide et l'offset au sein du bloc est aussi valide, et que la valeur v est lue à la case mémoire voulue et que les $n-1$ cases mémoire suivantes ont toutes le tag cont, alors l'appel renvoie Some v.

De façon analogue, un appel à store s'écrit : store *chunk mem block offset value*, où *chunk* est un chunk, *mem* une mémoire, *block* et *offset* des entiers relatifs représentant respectivement le bloc et l'offset où écrire la valeur, et *value* la valeur à écrire. Le type de retour est option mem :

- Si le bloc n'est pas valide (i.e n'a pas encore été alloué), ou si le bloc est valide mais que l'offset (ou celui de l'une des $n-1$ cases mémoire suivantes) n'est pas valide (i.e. est en dehors de l'intervalle de validité du bloc), alors l'appel renvoie None.
- Si le bloc est valide et l'offset au sein du bloc est aussi valide, et que les $n-1$ cases mémoire suivantes sont valides, alors l'appel renvoie Some m', où m' est la nouvelle mémoire modifiée.

Le modèle mémoire prévoit alors les théorèmes régissant la lecture et l'écriture de données :

- si l'écriture fonctionne, alors la lecture au même endroit renvoie la valeur écrite
- si l'écriture fonctionne, alors la lecture à un bloc différent est inchangée par l'écriture
- la lecture et l'écriture ne changent pas la validité des blocs ni des offsets au sein de chaque bloc
- l'allocation ne change pas les valeurs stockées dans les blocs autres que le bloc nouvellement alloué
- la libération ne change pas les valeurs stockées dans les blocs autres que le bloc libéré

Détails de la formalisation du contenu des cases mémoire Le type contentmap modélise le contenu d'un bloc : une fonction qui à chaque offset associe le contenu de la case mémoire ayant cet offset au sein du bloc. Ce contenu peut être de l'une des trois sortes suivantes :

- Undef, contenu initial non défini,
- Datum, une valeur accompagnée du nombre de cases mémoire devant contenir le tag cont,
- Cont, le tag cont.

```
Inductive content : Set :=
| Undef (* contenu initial non défini *)
| Datum : nat →val →content
| Cont.
Definition contentmap : Set := Z → content.
```

FIG. 11 – Formalisation du contenu d'un bloc

Au sein du bloc numéro *block* de la mémoire *mem*, une case mémoire est repérée par son offset o et son contenu est alors contents (blocks *mem block*) o, de type content.

Si les trois conditions suivantes sont réunies :

- la case mémoire d'offset o est valide et son contenu est Datum n v
- le *chunk* correspond à $n+1$ octets et est cohérent avec la valeur v
- les cases mémoire d'offset $o+1, \ldots, o+(\mathsf{Z\_of\_nat}\, n)$ suivantes sont valides et leur contenu est Cont

Alors, load *chunk mem block o* renvoie Some v.

À l'inverse, dans le cas d'une écriture, si les cases mémoire d'offset $o+1, \ldots, o+(\mathsf{Z\_of\_nat}\, n)$ suivantes sont valides, et si le *chunk* correspond à $n+1$ octets et est cohérent avec la valeur v, alors store *chunk mem block o v* renvoie Some *mem'* où le bloc numéro *block* de la nouvelle mémoire *mem'* est mis à jour, l'offset o recevant Datum n v et les offsets $o+1, \ldots, o+(\mathsf{Z\_of\_nat}\, n)$ recevant Cont, le reste étant inchangé.

1.4 Le langage Cminor

Le langage Cminor est le langage intermédiaire du compilateur CompCert, compilé vers l'assembleur PowerPC [Ler06], et vers lequel Clight, un sous-ensemble de C, est compilé [BDL06].

C'est un langage *C-like* : il possède des fonctionnalités de gestion mémoire bas niveau comme l'allocation de blocs mémoire, ou la lecture/écriture de valeurs en mémoire via des pointeurs. C'est un langage impératif structuré en procédures dont le corps est constitué d'instructions construites à partir d'expressions. Lors de l'appel d'une procédure, une pile d'exécution est fournie, mais les variables locales n'y sont pas allouées, celles-ci sont abstraites au niveau d'un environnement, et ne figurent pas dans la mémoire (en particulier, on ne peut pas construire de pointeur vers elles).

1.4.1 Syntaxe

Le langage Cminor est un langage impératif dont les *instructions* sont construites à partir d'*expressions*. Les instructions sont ensuite regroupées en *fonctions* qui peuvent avoir des *variables locales*. Un *programme* est un ensemble de fonctions et de *variables globales*.

Dans les grammaires syntaxiques ci-dessous, on prendra les conventions lexicales suivantes[2] :

[2]En réalité, en syntaxe concrète, tous les entiers considérés sont des entiers 32 bits. Mais en syntaxe abstraite, certaines constructions requièrent bel et bien des entiers naturels, de type nat en COQ.

ident	Identificateur (suite de caractères alphanumériques)
int	Entier 32 bits
float	Flottant
nat	Entier naturel
op1	Opérateur unaire (-, not, etc.)
op2	Opérateur binaire (+, -, /, etc.)

Expressions Une *expression* peut être[3] :

expr	::=	*int*	Constante entière
		float	Constante flottante
		op1 expr	Opérateur unaire
		expr op2 expr	Opérateur binaire
		expr?*expr*:*expr*	Expression conditionnelle
		"*ident*"	Variable globale
		ident	Variable locale
		$*nat*	Numéro de sous-expression (indice de De Bruijn)
		let *expr* in *expr*	Sous-expression (pour le partage d'expressions)
		&*int*	Pointeur de pile
		alloc *expr*	Allocation d'un bloc mémoire
		chunk[*expr*]	Lecture en mémoire
		chunk[*expr*] = *expr*	Écriture en mémoire
		expr(*expr, ...*) :*signature*	Appel d'une procédure

où, pour les *chunks* mémoire lors de la lecture et de l'écriture, on a :

chunk ::= | int8s | int8u | int16s | int16u | int32 | float32 | float64

correspondant en fait aux constructeurs M*chunk* (avec s pour signed et u pour unsigned) du type chunk.

Lors d'un appel de fonction, la *signature* est la donnée du type de la fonction appelée, information de typage de premier ordre (arguments non fonctionnels) :

type_arg ::= | int | float
type retour::= | int | float | void
signature ::= | *type_retour* | *type_arg*→*signature*

Un appel de fonction alloue une *pile* d'exécution, c'est à celle-ci que se réfère le pointeur de pile.

Notons que les expressions peuvent inclure des écritures mémoire, voire des allocations de blocs mémoire, mais pas d'assignations de variables locales. En outre, les variables de sous-expressions, une fois définies par let, ne peuvent pas être modifiées.

Instructions Une *instruction* peut être[4] :

[3] En réalité, il existe une classe spéciale d'expressions pour les conditions. Pour simplifier, on considérera qu'il s'agit d'expressions quelconques, qui doivent s'évaluer à l'entier 32 bits 0 ou 1 selon que la condition soit respectivement fausse ou vraie.

[4] La syntaxe présentée ici est simplifiée. En réalité, toutes les instructions sauf celles terminant par } doivent être suivies de ; . De plus, il existe des constructions raccourcies telles que le if sans else, ou le if à une seule instruction...

instr	::=	*expr*	Expression
	|	if (*expr*) {*instr*} else {*instr*}	Test
	|	*ident*=*expr*	Assignation d'une variable locale
	|	*instr* ; *instr*	Séquence d'instructions
	|	loop {*instr*}	Boucle
	|	{{*instr*}}	Bloc d'instructions
	|	exit *nat*	Sortie de *nat*+1 blocs
	|	return *expr*	Sortie d'une procédure avec une valeur de retour

On remarque qu'il n'existe aucune possibilité de *libérer* un bloc mémoire alloué auparavant.

Il n'y a pas de construction let pour les instructions : les variables de sous-expression ne peuvent pas « traverser » plusieurs instructions, elles sont cantonnées à une expression.

Procédures Une *procédure* est alors déclarée sous la forme suivante :

"*ident*" (*ident*, ...) : *signature* {
stack *int*
var *ident*, ...
instr
}

Une procédure peut également être déclarée sous forme *externe*, c'est-à-dire dont le code doit être cherché ailleurs que dans le programme source Cminor. Typiquement, cette syntaxe est utilisée pour modéliser les appels système.

extern "*ident*" : *signature*

Dans les deux cas, la signature correspond à celle utilisée lors de l'appel de la procédure.

Programmes Un *programme* s'écrit alors sous la forme suivante :

var "*ident*"[*int*]
...
procédure
...

qui correspond à la déclaration de variables globales suivie de la déclaration des procédures. On remarquera que rien n'est prévu *a priori* pour modifier les variables globales.

1.4.2 Sémantique

Les expressions s'*évaluent*, et les instructions s'*exécutent*, en modifiant éventuellement au passage la mémoire. Les expressions s'évaluent vers une valeur (de type val), tandis que les instructions ne donnent pas de résultat (autre que la mémoire modifiée).

L'exécution du programme correspond à l'appel de la procédure dont le nom est main. La valeur de retour de cette procédure doit être un entier 32 bits. Ce sera le code de retour du programme.

Trace d'exécution pour les appels externes Les événements tels que les entrées-sorties ne sont pas gérés par Cminor directement, mais par les procédures externes. Pour les intégrer à la sémantique du programme, ils sont modélisés par la notion de *trace* générée par l'appel d'une fonction externe. Cette trace est rendue abstraite par la sémantique de Cminor, elle est réduite à la donnée de :

– un opérateur de *concaténation* de deux traces, noté $\star$, qu'on suppose associatif

– une *trace vide* qui est *élément neutre* pour la concaténation

ces deux données faisant de l'ensemble des traces un monoïde. Les traces sont uniquement générées par l'appel d'une fonction externe, et sont donc générées à la *sortie* de chaque expression et instruction, mais ne sont jamais nécessaires à l'*entrée* de l'évaluation d'une expression ou de l'exécution d'une instruction. Un programme rend alors, en plus de son code de sortie, une trace d'exécution correspondant à la succession de tous ses appels externes.

Naïvement, on aurait pu utiliser le type des listes, avec l'opérateur de concaténation de listes ayant pour neutre la liste vide ; cependant, cela impliquerait que toutes les traces soient finies.

Alors, pour traiter la sémantique des appels de fonction externes, on se donne un oracle, extcall, qui prend en argument le nom de la fonction externe, les arguments *et la valeur de retour* et renvoie la trace correspondant à l'appel de la fonction. Ainsi, le nom et les arguments seuls ne permettent pas de déduire la valeur de retour de la fonction. En fait, un appel de fonction externe peut même renvoyer n'importe quelle valeur de retour, mais si cette valeur est fixée, alors une seule trace d'exécution convient.

Interprétation des variables Il y a trois sortes de variables :

– les variables globales pour tout le programme,
– les variables locales à chaque procédure
– et les variables de partage de sous-expression, au sein des expressions.

Pour interpréter les variables, il y a donc trois niveaux d'environnement.

Les variables globales ainsi que les déclarations de procédures réfèrent à un environnement global à tout le programme. Cet environnement ne leur associe pas une valeur, mais un bloc, celui-ci ne pouvant être modifié. Les valeurs proprement dites des variables globales sont donc directement dans la mémoire.

les variables locales réfèrent à un environnement local à une procédure. Cet environnement leur associe une valeur, qui peut être modifiée par une instruction d'affectation ; ces valeurs ne sont pas stockées dans la mémoire : on ne peut donc pas construire de « pointeur vers une variable locale ». Un environnement de variables locales sera alors une fonction de type ident→option val telle qu'une variable est associée à None si et seulement si elle n'est pas déclarée. On notera nullenv l'environnement de variables locales associant None à toutes les variables. Un environnement *env* modifié sera noté $env[var \leftarrow optionval]$ où la variable *var* est associée à l'option valeur *optionval*, toutes les autres inchangées par ailleurs.

les variables de partage de sous-expression (construction let) réfèrent à un environnement local à une instruction. (En effet, une construction let ne peut pas traverser plusieurs instructions, elle est cantonnée à une expression). Cet environnement leur associe une valeur qui ne peut pas être modifiée. En fait, cet environnement peut être vu comme une pile, telle que chaque construction let *expr1* in *expr2* empile la valeur de *expr1* pour l'interprétation de *expr2*, cette valeur étant ensuite dépilée ; la variable de partage de sous-expression est alors un numéro indiquant à quelle profondeur de la pile chercher la valeur de la sous-expression. Les valeurs des sous-expressions ne sont pas non plus stockées dans la mémoire : on ne peut donc pas construire de « pointeur vers une variable de partage de sous-expression ».

Construction de l'environnement global et de la mémoire initiale La construction de l'environnement global s'effectue à partir des déclarations des variables globales ainsi que des procédures. Pour chaque variable globale, un bloc mémoire est alloué statiquement, sa taille étant indiquée dans la déclaration de la variable globale. En même temps que l'on construit l'environnement global, on obtient donc également l'état mémoire initial avant exécution du programme.

Mais ceci est également vrai pour les procédures : en fait, l'environnement global associe chaque variable globale et chaque identificateur de procédure à un bloc, et chaque bloc correspondant à une procédure est associé à sa procédure. Ceci permet en fait de manipuler des « pointeurs vers des procédures ». Cependant, en toute rigueur, il faudrait prouver que ces blocs demeurent inchangés au cours

de l'exécution du programme, même si la mémoire est modifiée. En pratique c'est le cas, l'astuce étant que les numéros de blocs associés aux fonctions sont négatifs, et que, avant l'exécution du programme, l'état mémoire initial est tel que tous les blocs négatifs sont de taille nulle, donc aucune écriture n'y est possible.

Ainsi, une variable globale dans une expression Cminor peut référer tout aussi bien à une variable globale du programme qu'à une procédure, interne ou externe.

Pour simplifier, notons donc *genv* l'environnement global, qu'on prendra de type ident→option block, et *gfuncts* l'environnement des fonctions, qu'on prendra de type block→option func, où func est le type représentant les déclarations de procédures en syntaxe abstraite.

- Si var "*ident*"[*int*] est une déclaration de variable globale, alors *genv ident*, existera et sera le numéro, positif, de son bloc, bloc dont les offsets valides seront compris entre zéro inclus et signed *int* exclu[5]
- Si on a la déclaration de procédure p, interne ou externe, définissant une procédure de nom *ident*, alors *genv ident* = Some n, où n est le numéro, négatif, du bloc de la procédure, dont aucun offset ne sera valide. Et alors, *gfuncts* n = Some p.

Ces environnements globaux sont construits une fois pour toutes avant l'exécution proprement dite du programme ; ils ne peuvent être modifiés. Ils seront donc fixés dans toute la suite.

Jugements d'évaluation des expressions et d'exécution des instructions L'écriture de valeurs dans une mémoire, ainsi que l'allocation, sont prévus syntaxiquement au niveau des expressions. Celles-ci modifient donc la mémoire. Cependant, la modification de variables locales est prévue uniquement au niveau des instructions. Les expressions ne modifient donc pas l'environnement des variables locales.

Fixons donc un environnement de variables locales *env*. Alors, si m est une mémoire, *lenv* une liste constituant la liste des variables de sous-expressions, et *sp* le bloc de la pile de la procédure où s'évalue l'expression, on notera :

$$env, sp, lenv \vdash \{mem\}\, expr \Rightarrow val\, \langle trace \rangle\, \{mem'\}$$

pour exprimer que l'expression *expr* s'évalue vers la valeur *val* en générant la trace *trace* et en modifiant la mémoire *mem* en la mémoire *mem'*. On notera $\langle\rangle$ si la trace est vide.

On définit alors une sémantique *à grands pas* pour les instructions : certes, elles ne renvoient pas de valeur, mais elles s'exécutent avec un *tag de retour* qui indique si l'exécution doit se poursuivre, ou si on sort d'un bloc, ou d'une procédure :

- Out_return *val* : on sort de la procédure avec la valeur de retour *val*
- Out_exit n : on sort de n+1 blocs
- Out_normal : l'exécution se poursuit normalement

Comme les instructions modifient l'environnement local, mais n'ont pas d'environnement de sous-expressions, on notera alors :

$$sp \vdash \{env, mem\}\, instr \rightsquigarrow outcome\, \langle trace \rangle\, \{env', mem'\}$$

pour exprimer que l'instruction *instr* s'exécute avec le tag de retour *outcome* en générant la trace *trace* et en modifiant la mémoire *mem* en la mémoire *mem'*.

[5] *int* étant un entier 32 bits, signed permet de calculer sa représentation signée, c'est-à-dire dans l'intervalle $\left[-2^{31}, 2^{31}-1\right]$ de $\mathbb{Z}$

Expressions Les constantes entières et flottantes s'évaluent vers les valeurs attendues :

$$\frac{}{env, sp, lenv \vdash \{mem\}\, int \Rightarrow \mathsf{Vint}\, int\, \langle\rangle\, \{mem\}}$$

$$\frac{}{env, sp, lenv \vdash \{mem\}\, float \Rightarrow \mathsf{Vfloat}\, float\, \langle\rangle\, \{mem\}}$$

Les opérateurs unaires et binaires sont des opérateurs arithmétiques ou booléens sur les entiers, les flottants et les pointeurs, donc ils ne modifient pas la mémoire, et ne génèrent aucune trace. Leur sémantique sera sous-entendue ici. Pour les opérateurs booléens ainsi que les opérateurs de condition, le booléen vrai (resp. faux) est représenté par l'entier 32 bits 1 (resp. 0). Cependant, un détail technique assez important par la suite est que la comparaison de pointeurs ne s'effectue que si les pointeurs sont valides, c'est-à-dire que les blocs vers lesquels ils pointent sont valides, et que les offsets désignés par les pointeurs sont valides au sein de ces blocs.

L'expression conditionnelle s'évalue d'abord par la condition, qui doit s'évaluer en un entier 0 (faux) ou 1 (vrai). Alors, en fonction de cet entier, la branche correspondante de l'expression est évaluée.

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, exprtest \Rightarrow \mathsf{Vint}\, \mathsf{one}\, \langle tr\rangle\, \{mem''\} \\ env, sp, lenv \vdash \{mem''\}\, expr_1 \Rightarrow v\, \langle tr_1\rangle\, \{mem'\} \end{array}}{env, sp, lenv \vdash \{mem\}\, exprtest\, ?expr_1\, :expr_0 \Rightarrow v\, \langle tr \star tr_1\rangle\, \{mem'\}}$$

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, exprtest \Rightarrow \mathsf{Vint}\, \mathsf{zero}\, \langle tr\rangle\, \{mem''\} \\ env, sp, lenv \vdash \{mem''\}\, expr_0 \Rightarrow v\, \langle tr_0\rangle\, \{mem'\} \end{array}}{env, sp, lenv \vdash \{mem\}\, exprtest\, ?expr_1\, :expr_0 \Rightarrow v\, \langle tr \star tr_0\rangle\, \{mem'\}}$$

Une variable globale s'évalue en faisant appel à l'environnement *genv*. Mais elle s'évalue en le pointeur vers son bloc, car sa valeur proprement dite est stockée en mémoire. L'offset de ce pointeur est zéro. Un nom de procédure interne ou externe peut être également pris comme une variable globale : il s'évaluera alors en le pointeur vers son bloc de procédure.

$$\frac{genv\, ident = \mathsf{Some}\, b}{env, sp, lenv \vdash \{mem\}\, \text{"}ident\text{"} \Rightarrow \mathsf{Vptr}\, b\, \mathsf{zero}\, \langle\rangle\, \{mem\}}$$

Une variable locale s'évalue en faisant appel à l'environnement *env* :

$$\frac{env\, ident = \mathsf{Some}\, v}{env, sp, lenv \vdash \{mem\}\, ident \Rightarrow v\, \langle\rangle\, \{mem\}}$$

Une variable de sous-expression s'évalue en faisant appel à l'environnement *lenv*, qui est en fait une pile, ou une liste ; la variable est un entier naturel qui indique à quelle position de la liste/pile lire la valeur. C'est le principe des indices de De Bruijn.

$$\frac{}{env, sp, (v :: lenv) \vdash \{mem\}\, \$0 \Rightarrow v\, \langle\rangle\, \{mem\}}$$

$$\frac{env, sp, lenv \vdash \{mem\}\, \$n \Rightarrow v\, \langle\rangle\, \{mem\}}{env, sp, (v' :: lenv) \vdash \{mem\}\, \$\mathsf{S}n \Rightarrow v\, \langle\rangle\, \{mem\}}$$

La construction let permet de partager une sous-expression en empilant sa valeur dans l'environnement des sous-expressions. Attention, cette construction substitue la *valeur* de la sous-expression et non la sous-expression elle-même. Elle évalue donc la sous-expression une fois pour toutes.

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, expr_0 \Rightarrow v_0\, \langle tr_0 \rangle\, \{mem_0\} \\ env, sp, (v_0 :: lenv) \vdash \{mem_0\}\, expr \Rightarrow v\, \langle tr \rangle\, \{mem'\} \end{array}}{env, sp, lenv \vdash \{mem\}\, \mathsf{let}\, expr_0\, \mathsf{in}\, expr \Rightarrow v\, \langle tr_0 \star tr \rangle\, \{mem'\}}$$

Un pointeur de pile s'évalue comme un pointeur dont le bloc est le bloc de pile sp, et l'offset l'entier spécifié.

$$\frac{}{env, sp, lenv \vdash \{mem\}\, \&int \Rightarrow \mathsf{Vptr}\, sp\, int\, \langle\rangle\, \{mem\}}$$

L'allocation d'un bloc mémoire est une expression. Un bloc de bornes 0 et l'entier spécifié est alloué avec la fonction alloc du modèle mémoire, et un pointeur vers le début de ce bloc est renvoyé.

$$\frac{\mathsf{alloc}\, mem\, 0\, (\mathsf{signed}\, int) = (mem', b)}{env, sp, lenv \vdash \{mem\}\, \mathsf{alloc}\, int \Rightarrow \mathsf{Vptr}\, b\, \mathsf{zero}\, \langle\rangle\, \{mem'\}}$$

La lecture mémoire s'évalue alors en utilisant la fonction load du modèle mémoire. Attention, la lecture s'effectue après évaluation de l'expression qui doit renvoyer le pointeur où lire.

$$\frac{env, sp, lenv \vdash \{mem\}\, expr \Rightarrow \mathsf{Vptr}\, b\, i\, \langle tr \rangle\, \{mem'\} \quad \mathsf{load}\, \mathsf{M}chunk\, mem'\, b\, (\mathsf{signed}\, i) = \mathsf{Some}\, v}{env, sp, lenv \vdash \{mem\}\, chunk\, [expr] \Rightarrow v\, \langle tr \rangle\, \{mem'\}}$$

L'écriture mémoire s'évalue alors en utilisant la fonction store du modèle mémoire. Attention, l'écriture s'effectue après évaluation d'abord de l'expression qui doit renvoyer le pointeur où écrire, puis de l'expression qui doit renvoyer la valeur à écrire. Comme en C, la valeur renvoyée est la valeur écrite, ce qui permet d'enchaîner des écritures.

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, expr_1 \Rightarrow \mathsf{Vptr}\, b\, i\, \langle tr_1 \rangle\, \{mem_1\} \\ env, sp, lenv \vdash \{mem_1\}\, expr_2 \Rightarrow v\, \langle tr_2 \rangle\, \{mem_2\} \\ \mathsf{store}\, \mathsf{M}chunk\, mem_2\, b\, (\mathsf{signed}\, i)\, v = \mathsf{Some}\, mem' \end{array}}{env, sp, lenv \vdash \{mem\}\, chunk\, [expr_1] = expr_2 \Rightarrow v\, \langle tr_1 \star tr_2 \rangle\, \{mem'\}}$$

L'appel de procédure s'effectue en trois étapes. D'abord, il faut déterminer quelle procédure utiliser. Cette procédure n'est pas nécessairement donnée par son nom, mais par une expression quelconque, qui doit donc s'évaluer vers un pointeur qui, par son bloc, donnera la procédure à utiliser. Puis, les arguments sont évalués un par un successivement, en partant de la gauche, la mémoire pouvant être modifiée au fur et à mesure. Enfin vient l'exécution du corps de la procédure. Cette étape se divise en deux cas : le cas interne, et le cas externe. Soit f un terme Coq représentant la déclaration de la procédure à utiliser.

Dans le cas interne, f s'écrit :

"*procname*" (*arg1*, ...) : *sig* {
stack *stacksize*
var *local1*, ...
instr ...
}

Alors, une fois les arguments évalués, un bloc de pile est alloué, et l'environnement local est initialisé avec les valeurs des arguments et une valeur non définie pour les variables locales. Après l'exécution du corps de la procédure, le bloc de pile est libéré. On a alors :

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, expr_0 \Rightarrow \mathsf{Vptr}\, b\, i\, \langle tr_0 \rangle\, \{mem_0\} \\ gfuncts\, b = f \\ \\ env, sp, lenv \vdash \{mem_0\}\, expr_1 \Rightarrow v_1\, \langle tr_1 \rangle\, \{mem_1\} \\ \vdots \\ env, sp, lenv \vdash \{mem_{n-1}\}\, expr_n \Rightarrow v_n\, \langle tr_n \rangle\, \{mem_n\} \\ \\ \mathsf{alloc}\, mem_n\, 0\, stacksize = \left(mem^{(1)}, sp'\right) \\ \\ env_0 = ((\ldots(\mathsf{nullenv}\, [arg_1 \leftarrow \mathsf{Some}\, v_1])\ldots)\, [arg_n \leftarrow \mathsf{Some}\, v_n]) \\ env_1 = ((\ldots(env_0\, [local_1 \leftarrow \mathsf{Some\,Vundef}])\ldots)\, [local_n \leftarrow \mathsf{Some\,Vundef}]) \\ \\ sp' \vdash \left\{env_1, mem^{(1)}\right\}\, instr \rightsquigarrow \mathsf{Out\_return}\, v\, \langle tr' \rangle\, \left\{env_2, mem^{(2)}\right\} \\ \\ \mathsf{free}\, mem^{(2)}\, sp' = mem' \end{array}}{env, sp, lenv \vdash \{mem\}\, expr_0\, (expr_1, \ldots, expr_n) : sig \Rightarrow v\, \langle tr_0 \star tr_1 \star \ldots \star tr_n \star tr' \rangle\, \{mem'\}}$$

Dans le cas des appels externes, si f s'écrit :

extern "*procname*" : *sig*

Alors on utilise l'oracle extcall avec les arguments et la valeur de retour, pour trouver la trace.

$$\frac{\begin{array}{c} env, sp, lenv \vdash \{mem\}\, expr_0 \Rightarrow \mathsf{Vptr}\, b\, i\, \langle tr_0 \rangle\, \{mem_0\} \\ gfuncts\, b = f \\ \\ env, sp, lenv \vdash \{mem_0\}\, expr_1 \Rightarrow v_1\, \langle tr_1 \rangle\, \{mem_1\} \\ \vdots \\ env, sp, lenv \vdash \{mem_{n-1}\}\, expr_n \Rightarrow v_n\, \langle tr_n \rangle\, \{mem_n\} \\ \\ \mathsf{extcall}\, procname\ (v_1 :: \ldots :: v_n :: \mathsf{nil})\ v = tr' \end{array}}{env, sp, lenv \vdash \{mem\}\, expr_0\, (expr_1, \ldots, expr_n) : sig \Rightarrow v\, \langle tr_0 \star tr_1 \star \ldots \star tr_n \star tr' \rangle\, \{mem\}}$$

Il est intéressant de noter que dans ce cas, il n'y a pas d'allocation de bloc de pile. Plus précisément, une fois la procédure déterminée et les arguments évalués, l'appel externe lui-même ne modifie pas la mémoire.

Instructions Une expression peut être exécutée en tant qu'instruction. C'est souvent le cas par exemple, pour les écritures en mémoire, où on n'a pas toujours immédiatement besoin de la valeur écrite, ou si, par exemple, c'est une constante qu'on écrit en mémoire. Dans ce cas, l'expression est évaluée dans un environnement de sous-expressions vide, et le résultat de l'évaluation de l'expression est oublié, mais pas la trace.

$$\frac{env, sp, \mathsf{nil} \vdash \{mem\}\, expr \Rightarrow v\, \langle tr \rangle\, \{mem'\}}{sp \vdash \{env, mem\}\, expr \rightsquigarrow \mathsf{Out\_normal}\, \langle tr \rangle\, \{env, mem'\}}$$

Un test joue le rôle pour les instructions d'une expression conditionnelle. L'expression représentant la condition est évaluée dans un environnement de sous-expressions vide.

$$\frac{\begin{array}{c} env, sp, \mathsf{nil} \vdash \{mem\}\, exprtest \Rightarrow \mathsf{Vint\ one}\, \langle tr \rangle\, \{mem''\} \\ sp \vdash \{env, mem''\}\, instr_1 \rightsquigarrow out\, \langle tr_1 \rangle\, \{env', mem'\} \end{array}}{sp \vdash \{env, mem\}\, \mathsf{if}\, (exprtest)\, \{instr_1\}\, \mathsf{else}\, \{instr_0\} \rightsquigarrow out\, \langle tr \star tr_1 \rangle\, \{env', mem'\}}$$

$$\frac{\begin{array}{c} env, sp, \mathsf{nil} \vdash \{mem\}\, exprtest \Rightarrow \mathsf{Vint\ zero}\, \langle tr \rangle\, \{mem''\} \\ sp \vdash \{env, mem''\}\, instr_0 \rightsquigarrow out\, \langle tr_0 \rangle\, \{env', mem'\} \end{array}}{sp \vdash \{env, mem\}\, \mathsf{if}\, (exprtest)\, \{instr_1\}\, \mathsf{else}\, \{instr_0\} \rightsquigarrow out\, \langle tr \star tr_0 \rangle\, \{env', mem'\}}$$

L'assignation d'une variable locale est prévue non pas au niveau des expressions, mais des instructions. Elle se déroule comme l'évaluation d'une expression en tant qu'instruction, sauf que la valeur de l'expression n'est pas oubliée. Mais elle ne se déroule qu'à condition que la variable locale soit déclarée.

$$\frac{\begin{array}{c} env, sp, \mathsf{nil} \vdash \{mem\}\, expr \Rightarrow v\, \langle tr \rangle\, \{mem'\} \\ env\, ident = \mathsf{Some}\, v_0 \end{array}}{sp \vdash \{env, mem\}\, ident = expr \rightsquigarrow \mathsf{Out\_normal}\, \langle tr \rangle\, \{env\, [ident \leftarrow \mathsf{Some}\, v]\, , mem'\}}$$

Deux instructions peuvent se succéder. Elles s'exécutent alors l'une à la suite de l'autre, sauf si la première est interrompue, auquel cas la seconde n'est pas exécutée du tout et le tag de retour est celui de la première instruction.

$$\frac{\begin{array}{c} sp \vdash \{env, mem\}\, instr_1 \rightsquigarrow \mathsf{Out\_normal}\, \langle tr_1 \rangle\, \{env_1, mem_1\} \\ sp \vdash \{env_1, mem_1\}\, instr_2 \rightsquigarrow out\, \langle tr_2 \rangle\, \{env', mem'\} \end{array}}{sp \vdash \{env, mem\}\, instr_1; instr_2 \rightsquigarrow out\, \langle tr_1 \star tr_2 \rangle\, \{env', mem'\}}$$

$$\frac{\begin{array}{c} sp \vdash \{env, mem\}\, instr_1 \rightsquigarrow out\, \langle tr \rangle\, \{env', mem'\} \\ out \neq \mathsf{Out\_normal} \end{array}}{sp \vdash \{env, mem\}\, instr_1; instr_2 \rightsquigarrow out\, \langle tr \rangle\, \{env', mem'\}}$$

Une boucle est la répétition d'une instruction tant qu'elle se termine normalement. Si elle est interrompue, alors la boucle s'arrête avec le même tag de retour.

$$\frac{\begin{array}{c} sp \vdash \{env, mem\}\, instr \rightsquigarrow \mathsf{Out\_normal}\, \langle tr_1 \rangle\, \{env_1, mem_1\} \\ sp \vdash \{env_1, mem_1\}\, \mathsf{loop}\, instr \rightsquigarrow out\, \langle tr_2 \rangle\, \{env', mem'\} \end{array}}{sp \vdash \{env, mem\}\, \mathsf{loop}\, instr \rightsquigarrow out\, \langle tr_1 \star tr_2 \rangle\, \{env', mem'\}}$$

$$\frac{\begin{array}{c} sp \vdash \{env, mem\}\, instr_1 \rightsquigarrow out\, \langle tr \rangle\, \{env', mem'\} \\ out \neq \mathsf{Out\_normal} \end{array}}{sp \vdash \{env, mem\}\, \mathsf{loop}\, instr \rightsquigarrow out\, \langle tr \rangle\, \{env', mem'\}}$$

On notera qu'aucune interruption *spécifique* d'une boucle n'est prévue dans la syntaxe. L'idée pour interrompre une boucle est alors de l'inclure dans un bloc. Alors, sortir du bloc impliquera l'interruption de la boucle.

Un bloc est donc une instruction qu'on peut interrompre (typiquement, une boucle). Un tag Out_exit indique le nombre de blocs dont il faut sortir.

$$\frac{sp \vdash \{env, mem\}\, instr \rightsquigarrow \mathsf{Out\_normal}\, \langle tr \rangle\, \{env', mem'\}}{sp \vdash \{env, mem\}\, \{\{instr\}\} \rightsquigarrow \mathsf{Out\_normal}\, \langle tr \rangle\, \{env', mem'\}}$$

$$\frac{sp \vdash \{env, mem\}\, instr \rightsquigarrow \mathsf{Out\_exit}\ 0\ \langle tr\rangle\ \{env', mem'\}}{sp \vdash \{env, mem\}\, \{\{instr\}\} \rightsquigarrow \mathsf{Out\_normal}\ \langle tr\rangle\ \{env', mem'\}}$$

$$\frac{sp \vdash \{env, mem\}\, instr \rightsquigarrow \mathsf{Out\_exit}\ (\mathsf{S}n)\ \langle tr\rangle\ \{env', mem'\}}{sp \vdash \{env, mem\}\, \{\{instr\}\} \rightsquigarrow \mathsf{Out\_exit}\ n\ \langle tr\rangle\ \{env', mem'\}}$$

$$\frac{sp \vdash \{env, mem\}\, instr \rightsquigarrow \mathsf{Out\_return}\ v\ \langle tr\rangle\ \{env', mem'\}}{sp \vdash \{env, mem\}\, \{\{instr\}\} \rightsquigarrow \mathsf{Out\_return}\ v\ \langle tr\rangle\ \{env', mem'\}}$$

Les instructions permettant de sortir d'un bloc sont soit exit, spécifique aux blocs, soit return, qui sort de toute la procédure.

$$\frac{}{sp \vdash \{env, mem\}\, \mathsf{exit}\ nat \rightsquigarrow \mathsf{Out\_exit}\ nat\ \langle\rangle\ \{env, mem\}}$$

$$\frac{env, sp, \mathsf{nil} \vdash \{mem\}\, expr \Rightarrow v\ \langle tr\rangle\ \{mem'\}}{sp \vdash \{env, mem\}\, \mathsf{return}\ expr \rightsquigarrow \mathsf{Out\_return}\ v\ \langle tr\rangle\ \{env, mem'\}}$$

Programme Si *mem* est la mémoire initiale construite en même temps que les environnements globaux, alors la sémantique du programme est le couple (i, tr) du code de sortie et de la trace données par le jugement suivant (la mémoire *mem'* après exécution n'est pas prise en compte dans cette sémantique) :

$$\mathsf{nullenv}, 0, \mathsf{nil} \vdash \{mem\}\, \mathsf{main}\ () : \mathsf{int} \Rightarrow \mathsf{Vint}\ i\ \langle tr\rangle\ \{mem'\}$$

Ce couple n'est pas unique à cause des appels de procédures externes. Ainsi, la sémantique de Cminor n'est pas déterministe.

1.5 Aperçu du stage

Au cours de ce stage, j'ai réalisé les travaux suivants.

J'ai tout d'abord formalisé la structure de la mémoire : tas, racines, notion de chemin dans la mémoire. Ceci correspond au module Memory. Le module Memory_abstract est préalablement nécessaire pour définir une notion abstraite d'objet utilisé.

J'ai ensuite considéré un ramasse-miettes de type *mark and sweep* avec coalescence. Xavier Leroy en a proposé une implémentation en Cminor, et c'est à partir de ces fragments de code, dont j'ai précisé ou corrigé certains points, que j'ai amorcé la preuve de correction du GC. À ce jour, j'ai réussi à prouver l'étape mark, que je détaillerai au chapitre 3. Mais les preuves du sweep et de l'allocateur, vues au chapitre 4, demeurent inachevées. Ceci est essentiellement dû au fait que dans le cas de l'allocation et de la coalescence, la structure de la mémoire est modifiée, et les preuves se révèlent donc beaucoup plus difficiles que pour l'étape mark.

La preuve du GC est organisée en plusieurs modules :

– Marksweep_concrete : définition des programmes en Cminor et des algorithmes correspondants écrits directement en Coq (langage de spécification *Gallina*).

- Marksweep_concrete_cminor : les programmes en Cminor et les algorithmes effectuent les mêmes tâches et donnent le même résultat
- Marksweep_concrete_proof : preuve de correction et de terminaison des algorithmes de mark and sweep
- Mark_abstract : preuve de correction des algorithmes *abstraits* de marquage, c'est-à-dire, en supposant que la structure de la mémoire soit conservée
- Mark_abstract_finite : preuve de terminaison des algorithmes *abstraits* de marquage sous l'hypothèse du tas fini

Il faut noter que l'allocateur est largement indépendant du GC. C'est pourquoi aucun de ces modules ne traite de l'allocation : la preuve de l'allocateur et isolée dans le module Memalloc.

Le traitement des entiers 32 bits a également posé quelques difficultés. En l'absence d'une tactique automatique telle que omega, j'ai donc dû développer des librairies supplémentaires pour raisonner sur les entiers 32 bits, mais aussi les listes et les boucles. Ces librairies (Int_addons, List_addons, Loop) seront présentées en annexe.

Les sources Coq des modules du stage seront disponibles prochainement sur le site [Ram07]. La compilation de ces sources requiert celle de tout le compilateur certifié CompCert ; or seules les spécifications de CompCert sont disponibles au public (les preuves elles-mêmes ne sont que partiellement disponibles). Merci d'écrire à Xavier Leroy[6] pour de plus amples renseignements.

[6] Xavier.Leroy@inria.fr

2 Structure de la mémoire pour sa gestion haut niveau

La structure de la mémoire est définie à trois niveaux :

– les objets sont organisés sous la forme d'un tas
– chaque objet a des objets fils vers lesquels il pointe
– des pointeurs vers les objets racine sont fournis

Au moyen de cette structure, on peut alors définir la notion de *chemin* dans la mémoire, représentée comme une forêt dont les racines sont justement les objets racine.

Cette formalisation constitue une part importante de ce stage, au sein du module Memory.

2.1 Le tas

2.1.1 Conception

On pourrait penser naïvement que chaque objet occupe son propre bloc mémoire. Dans ce cas, on pourrait utiliser directement l'allocation de bloc mémoire en guise d'allocateur. Dans ce modèle mémoire, toutes les allocations réussiraient, la mémoire ne serait jamais épuisée et on n'aurait même pas besoin de GC.

Cependant, bien évidemment, le problème de la mémoire finie ressurgirait au moment de passer à la mémoire concrète (avec un seul bloc mémoire et des adressages absolus). On pourrait alors ajouter un GC pour pallier ce problème. Mais il serait insuffisamment résolu, car l'espace libre ne serait pas compacté (le modèle mémoire ne prévoyant rien concernant l'agencement des blocs).

L'idée de stocker le tas dans un seul bloc permet d'implémenter un GC stop and copy qui puisse compacter l'espace libre. Bien que je n'aie pas traité ce type de GC dans ce stage, j'ai conservé ce choix d'implémentation du tas afin que sa structure soit indépendante du GC.

Les objets sont stockés contigûment sous le format suivant :

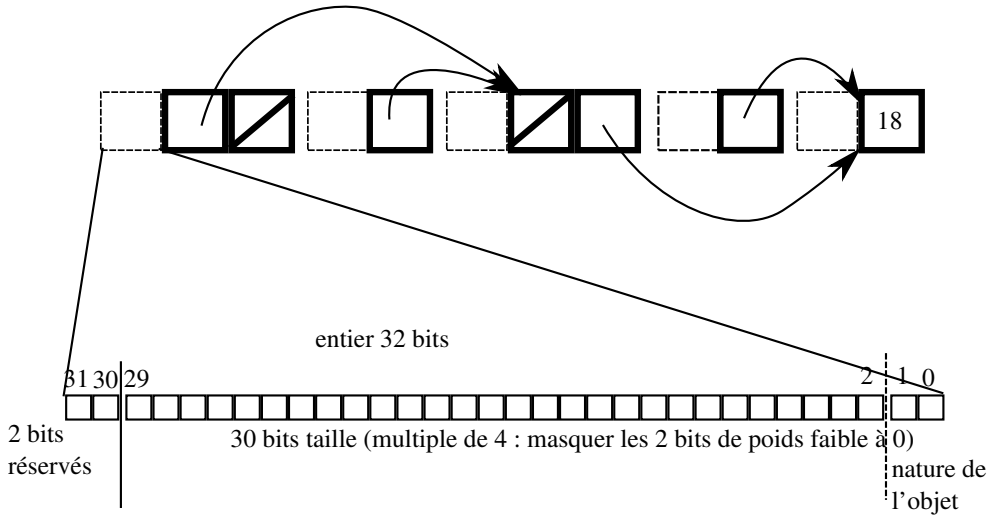

Fig. 12 – Structure du tas

1. Une en-tête constituée d'un entier de 32 bits. Les 2 bits de poids fort sont réservés à une utilisation future. Nous verrons plus loin qu'ils sont utilisés par le ramasse-miettes. Les

30 bits restants sont donc utilisés pour stocker la taille en octets de l'objet. Or, tous les accès lecture et écriture considérés dans ce stage sont des chunks `Mint32` de 4 octets. Donc, cette taille est un multiple de 4. Alors, les deux bits de poids faible peuvent être utilisés à un autre effet : pour lire la taille de l'objet, on masquera ces deux bits à 0. En fait, on utilisera ces deux bits pour indiquer la nature de l'objet. (Cf. tableau 13). On dispose donc de trois fonctions :

- les deux bits de poids fort sont lus via Extra_header
- les deux bits de poids faible sont lus via Kind_header
- la taille (c'est-à-dire les 30 bits de poids faible, dont les deux bits de poids faible sont masqués à zéro) est lue via Size_header

2. Les données elles-mêmes, sous la forme d'entiers de 32 bits ou de pointeurs, en fonction de la nature de l'objet. Dans le cas d'un objet de données (KIND_RAWDATA), ces données auraient pu être des valeurs quelconques, mais l'en-tête ne donne pas assez d'information sur la nature des valeurs stockées. Donc, pour simplifier, on choisit un chunk unique, Mint32 (correspondant donc à des entiers 32 bits ou des pointeurs).

Nom	Valeur des bits de poids faible	Nature des données	Description
KIND_RAWDATA	0	Entiers 32 bits	Données seulement
KIND_PTRDATA	1	Pointeurs	Structure de pointeurs
KIND_CLOSURE	2	Un entier 32 bits + pointeurs	Fermeture (l'entier indique le pointeur de code)

FIG. 13 – Natures possibles des objets

2.1.2 Tas bien formé : well_formed_from_to

Plus formellement, on exprime qu'un tas est bien formé, au moyen d'un prédicat inductif, well_formed_from_to.

```
Inductive well_formed_from_to (m : mem) (heap : block) : int → int → Prop :=
| well_formed_end : ∀ he, well_formed_from_to m heap he he
| well_formed_block : ∀ he n, (n = he → False) →
∀ v, Some (Vint v) = load Mint32 m heap (signed n) →
(Size_header v ≠ zero → ∀ x, modulo_four x n → cmp Cle (add n four) x = true →
cmp Cle x (add n (Size_header v)) = true →
valid_access m Mint32 heap (signed x)) →
cmp Cle n (add n three) = true →
cmp Cle (add n three) (add (add n (Size_header v)) three) = true →
cmp Cle (add (add n (Size_header v)) three) (sub he one) = true →
well_formed_from_to m heap (add (add n (Size_header v)) four) he →
well_formed_from_to m heap n he.
```

FIG. 14 – Le tas est bien formé (well_formed_from_to)

Soit m la mémoire considérée, *heap* le numéro du bloc mémoire contenant le tas, et n, he des entiers 32 bits représentant respectivement l'offset du début du tas (en fait, de l'endroit à partir

duquel on lit le reste du tas) et l'offset de fin du tas (en fait, 1 + l'offset de la dernière case du tas).

- Si ces deux offsets sont égaux, alors le tas est bien formé. Cela correspond au tas vide.
- Sinon, on est en présence d'un objet. Il doit remplir les conditions suivantes :
  - une valeur entière 32 bits doit être lue à cet endroit. Cet entier 32 bits constitue en fait l'*en-tête* de l'objet considéré.
  - Si la taille s lue dans l'en-tête est non nulle, alors les accès aux données (offsets lus 4 par 4 à partir de la case suivante et jusqu'à la taille des données) doivent être valides.
  - trois conditions sur les offsets 32 bits, assurant la contiguïté de l'en-tête et des données. Ces conditions, $n \leq n+3 \leq n+3+s \leq he-1$, sont exprimées au sens des entiers 32 bits et non au sens des entiers relatifs quelconques. L'ordre considéré est l'ordre sur les représentations signées.
  - le tas doit être bien formé après la fin des données de l'objet

Les offsets doivent être manipulés sous la forme d'entiers 32 bits car la donnée de l'offset dans un pointeur est un entier 32 bits.

Par conséquent, toutes les opérations arithmétiques (addition, soustraction) sur les offsets sont effectuées modulo 2^{32}. Si les conditions d'ordre sur les offsets n'étaient pas respectées, alors on pourrait avoir des structures de tas non souhaitées où des en-têtes d'objets seraient lues dans les données d'autres objets.

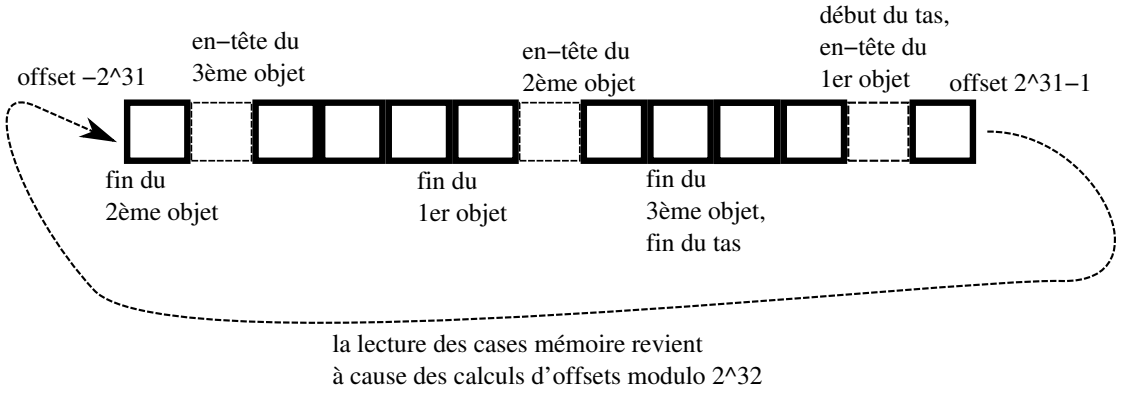

FIG. 15 – Un tas mal formé, où les conditions d'ordre sur les offsets ne sont pas respectées

Dans le cas d'un tas non vide ($n \neq he$), est important de noter que $he-1$ correspond à l'offset de la dernière case mémoire du tas, donc he désigne une case mémoire à l'*extérieur* du tas. On peut donc avoir par exemple que la dernière case du tas est « au bord », c'est-à-dire a l'offset maximal représentable en 32 bits signé : $he-1 = \mathsf{max\_signed}$ (l'entier 32 bits signé maximum, représentant $2^{31}-1$). Dans ce cas, $he = \mathsf{max\_signed}+1 = \mathsf{min\_signed}$ (l'entier 32 bits signé minimum, représentant -2^{31}). Et donc, si l'on avait écrit naïvement $n \leq n+4 \leq n+4+s \leq he$, alors par minimalité, on aurait $n = n+4 = n+4+s = he = \mathsf{min\_signed}$, en particulier $n = n+4$ ce qui est absurde. Mais si l'on avait également écrit naïvement $n \leq n+4 \leq n+4+s \leq he-1$, alors on aurait négligé le cas où $s = 0$ et l'objet est le dernier objet du tas (c'est-à-dire, que $n+3 = he-1$).

On remarquera que les deux branches du prédicat inductif s'excluent mutuellement, à cause de la comparaison entre n et he. Par conséquent, il est impossible d'avoir un tas qui occupe tout un espace de 2^{32} octets au sein du bloc. En effet, si c'était le cas, alors on aurait $n = he = \mathsf{min\_signed}$ (l'entier signé 32 bits minimum) et donc on ne pourrait pas le distinguer d'un tas vide.

2.1.3 Liste des offsets des objets : block_description_from_to

On peut adapter le prédicat well_formed_from_to en lui rajoutant un argument : une liste qui contiendra les offsets des objets. On obtient donc le prédicat block_description_from_to.

```
Inductive block_description_from_to (m : mem) (heap : block) : int → int → list int → Prop :=
| block_description_end : ∀ he, block_description_from_to m heap he he nil
| block_description_block : ∀ he n, (n = he → False) →
∀ v, Some (Vint v) = load Mint32 m heap (signed n) →
(Size_header v ≠ zero → ∀ x, modulo_four x n → cmp Cle (add n four) x = true →
cmp Cle x (add n (Size_header v)) = true →
valid_access m Mint32 heap (signed x)) →
cmp Cle n (add n three) = true →
cmp Cle (add n three) (add (add n (Size_header v)) three) = true →
cmp Cle (add (add n (Size_header v)) three) (sub he one) = true →
∀ l, block_description_from_to m heap (add (add n (Size_header v)) four) he →
block_description_from_to m heap n he (add n four :: l).
```

FIG. 16 – Liste des offsets des objets du tas (block_description_from_to)

Lorsqu'un objet est trouvé, le prédicat ajoute à la liste l'offset de son premier bloc de données, ou plus exactement 4 + l'offset de son en-tête. Il est important de souligner qu'il ne s'agit pas d'une liste de *pointeurs*, mais d'*offsets*, étant donné que tous les objets du tas se trouvent dans le même bloc.

Le prédicat block_description_from_to est fonctionnel : étant données une mémoire, un bloc mémoire et des bornes pour le tas, au plus une liste convient.

Pour chaque offset o présent dans la liste, on dispose de toutes les propriétés attendues pour l'objet qu'il représente :

- une valeur entière 32 bits est lue à l'offset $o-4$. Cet entier 32 bits constitue en fait l'*en-tête* de l'objet considéré.
- Si la taille s lue dans l'en-tête est non nulle, alors les accès aux données (offsets lus 4 par 4 à partir de la case o et jusqu'à $o+s-4$) sont valides.
- on a $o-4 \leq o-1 \leq o+s-1 \leq he-1$, pour assurer la contiguïté de l'en-tête et des données
- le tas est bien formé après la fin des données de l'objet (c'est-à-dire, à partir de l'offset $o+s$)

Ces lemmes se prouvent facilement par induction sur la liste (ou sur le prédicat block_description_from_to lui-même), ou bien en remarquant que si o est dans la liste, alors le tas est bien formé à partir de l'offset $o-4$.

On dispose également de lemmes supplémentaires dans le cas où la taille s est non nulle : on a alors que $o-1 < o \leq o+s-4 < o+s-1$ (au sens des entiers 32 bits). Ces lemmes découlent de théorèmes de la librairie des entiers 32 bits présentée en annexe.

Si on a un offset dans la liste, alors on peut s'en servir comme « pivot » pour diviser le tas en deux sous-tas : s'il existe deux listes l_1 et l_2 et un entier 32 bits b tels que
block_description_from_to m *heap* *hs* *he* $(l_1 +\!+ b :: l_2)$, alors on a
block_description_from_to m *heap* *hs* $(b-4)$ l_1 d'une part, et d'autre part
block_description_from_to m *heap* $(b-4)$ *he* $(b :: l_2)$.

Le fait d'introduire ces conditions sur les offsets des données permet de les manipuler plus facilement. En effet, ces conditions assurent qu'il est cohérent de considérer que, si la taille s

est non nulle, alors les données sont exactement l'ensemble des cases mémoire telles que leur offset est compris entre o et $o+s-4$ (au sens de l'ordre sur les représentations signées 32 bits) et congrues à o modulo 4. Si on étend l'intervalle à gauche en y ajoutant l'offset $o-4$, qui correspond à l'en-tête, alors on obtient toutes les cases mémoire de l'objet.

Mais ces conditions sur les offsets des données permettent également de montrer que la liste des offsets est triée selon l'ordre $\prec$ défini par $o_1 \prec o_2 :\Leftrightarrow o_1 - 4 < o_2 - 4$ au sens des représentations signées, c'est-à-dire triée dans l'ordre croissant des offsets des en-têtes. (Attention, $\prec$ n'est pas équivalent à $<$, l'ordre sur les offsets des objets eux-mêmes, car la soustraction est effectuée modulo 2^{32}.) Et alors, grâce à la librairie List_addons (décrite en annexe), on en déduit que :

- si o_1 et o_2 sont deux offsets distincts représentant des objets (c'est-à-dire qu'ils sont dans la liste des offsets des objets du tas), et que ces objets sont de tailles respectives s_1 et s_2, alors les intervalles $[o_1 - 4, o_1 + s_1 - 1]$ et $[o_2 - 4, o_2 + s_2 - 1]$ sont disjoints : deux objets distincts ne se chevauchent pas.
- si o_1 et o_2 sont deux offsets quelconques représentant des objets, et si la taille s_2 de l'objet o_2 est non nulle, alors $o_1 - 4$ n'est pas dans l'intervalle $[o_2, o_2 + s_2 - 1]$: l'en-tête d'un bloc n'est jamais dans une zone de données.

Si on dispose de deux tas adjacents (c'est-à-dire, tels que l'offset de fin d'un tas coïncide avec l'offset de début de l'autre) : s'il existe deux listes l_1 et l_2 et un entier 32 bits hm tels que block_description_from_to m *heap* *hs* *hm* l_1 d'une part, et d'autre part
block_description_from_to m *heap* *hm* *he* l_2, alors on a
block_description_from_to m *heap* *hs* *he* $(l_1 + +l_2)$, mais sous la condition suivante : $hs = hm \vee (hs \neq he \wedge hm - 1 \leq he - 1)$. En effet, dans le premier cas l_1 est vide, et dans le deuxième cas, la dernière case du tas l_1 doit être située avant la dernière case du tas l_2, mais la condition $hs \neq he$ doit être rajoutée afin de distinguer le tas fusionné du tas vide, ce cas ne pouvant être éliminé a priori.

2.1.4 Construction dans Set de la liste des offsets : exemple d'utilisation de la librairie Loop

Quelle est la condition telle qu'on puisse effectivement construire une telle liste ? Il est évident que block_description_from_to implique well_formed_from_to, on le vérifie en supprimant simplement la donnée de la liste dans la définition de block_description_from_to. La réciproque est vraie également : si well_formed_from_to est vérifiée, alors on peut trouver une liste.

Reste à savoir si on peut la construire dans **Set** (c'est-à-dire, pas seulement dans **Prop**). C'est effectivement le cas.

On peut essayer de le prouver directement, en utilisant la tactique refine :

```
refine (fun hs he l (H : well_formed_from_to m heap n he l) =>
match eq n he as eq_n_he return bool→_ with
| true => fun eq_n_he0 => _
| false => fun eq_n_he0 => _
end).
```

Le script de preuve se passe bien, mais au moment de le valider, avec la commande Defined, Coq occupe le temps processeur sans augmenter sa consommation mémoire, et sans bouger : Coq boucle. Il s'agit d'un bogue de type-checking qui est très difficile à reproduire sur un petit échantillon (donc, impossible d'envoyer un report de bogue à l'équipe de développement de Coq).

Il faut alors utiliser une structure de boucle générale. On utilise alors la librairie Loop, présentée en annexe. Elle formalise une boucle exécutée à partir d'un état (mémoire et environnement) de type A donné en paramètre. Le corps de la boucle est une fonction qui exécute une itération de la boucle, en prenant en argument un état de type A et en renvoyant l'état après l'itération, avec un booléen, false si la boucle doit être interrompue, true si une nouvelle itération avec le nouvel état doit être exécutée.

Dans notre cas, le corps de la boucle prend en entrée n (l'offset à partir duquel lire la suite du tas) et une liste l . Si $n = he$, alors la boucle s'arrête et la liste est renvoyée inchangée. Sinon, l'itération y ajoute à *droite* (et non à gauche avec cons) l'offset $n + 4$ de l'objet rencontré, et demande de continuer l'exécution de la boucle avec cette nouvelle liste et l'offset $n + s + 4$ où continuer de lire le tas.

```
Let body (m : mem) (heap : block) (he : int) (k : int × list int) :=
let (n, l) := k in
if eq_dec n he
then (false, (n, l))
else match load Mint32 m heap (signed n) with
| Some (Vint v) ⇒ let o := add (add n (Size_header v)) four in
(true, (o, (l ++ (add n four::nil))))
| _ ⇒ (false, k) (* cas absurde *)
end.
```

Alors, l'utilisation de la librairie Loop (présentée en annexe) définit un prédicat de terminaison (condition dans **Prop** nécessaire et suffisante pour que la boucle termine). Ce prédicat est montré par induction sur well_formed_from_to.

```
Lemma loop_term (m : mem) (heap : block) (he : int) :
∀ n, well_formed_from_to m heap n he → ∀ l, Loop.loop_term (body m heap he) (n, l).
```

La librairie Loop fournit alors le théorème loop pour montrer que si le prédicat de terminaison est vrai, alors on peut construire dans **Set** le résultat de l'exécution de la boucle.

```
Theorem loop (A : Set) (body : A → bool × A) :
∀ k : A, Loop.loop_term (A := A) body k →
{ k' : A | Loop.loop_prop (A := A) body k k' }.
```

Pour montrer la correction de la boucle, on montre d'abord que la boucle, si elle termine, renvoie une liste de la forme $l + +l''$, où l est la liste initiale et $++$ est la concaténation de listes.

```
Lemma loop_extensive (m : mem) (heap : block) (he : int) :
∀ a a', Loop.loop_prop (body m heap he) a a' →
∀ n l, a =(n, l) → ∀ n' l', a' = (n', l') →
∃ l'', l' = l ++ l''.
```

Et alors, grâce à ce lemme, on montre que l'' convient ; dans cette dernière preuve, on élimine, grâce à l'hypothèse well_formed_from_to, le cas absurde où $n \neq he$ mais qu'aucune valeur entière n'est lue à l'offset n :

```
Theorem loop_correct (m : mem) (heap : block) (he : int) :
∀ a a', Loop.loop_prop (body m heap he) a a' →
∀ n l, a =(n, l) → ∀ n' l', a' = (n', l') →
well_formed_from_to n p →
∀ l'', l' = l ++ l'' →
block_description_from_to n p l''.
```

Alors, il suffit d'exécuter la boucle avec $l = \mathsf{nil}$ initialement. L'égalité $l'' = \mathsf{nil} + + l''$ est en fait calculatoire, $a + + b$ étant défini par récurrence sur a.

2.1.5 Invariance de la structure de tas

Sous quelles conditions la structure de tas est-elle inchangée ? En fait, il suffit que les tailles lues dans les en-têtes ne changent pas.

Plus exactement, si m est une mémoire dont le tas est bien formé au bloc de numéro *heap*, avec l la liste des offsets représentant les objets, alors, si *m'* est une mémoire vérifiant les conditions suivantes pour chaque offset o présent dans l :

- une valeur entière est présente à l'offset $o - 4$ du bloc *heap'* de *m'*
- les tailles lues dans les valeurs entières à l'offset $o - 4$ des blocs *heap* de m et *heap'* de *m'* sont les mêmes
- la zone de données est valide dans le bloc *heap'* de *m'*

De ce théorème assez fort, on déduit des conditions plus fortes (donc, des théorèmes d'invariance plus faibles)

- si les en-têtes sont les mêmes, et si les zones de données sont valides dans *m'*
- ou si toute lecture pour tout offset et pour tout chunk donne les mêmes résultats dans le bloc *heap* de m que dans le bloc *heap'* de *m'* (ceci est vrai notamment dans le cas où *heap* = *heap'* et *m'* est le résultat de l'écriture d'une valeur dans un autre bloc, ou de l'allocation d'un nouveau bloc)

Mais la condition des mêmes tailles est également nécessaire : si deux mémoires ont la même liste d'offsets entre les mêmes bornes *hs* et *he*, alors tout objet d'offset o présent dans la liste admet la même taille dans les deux mémoires.

2.2 Les objets fils d'un objet

Rappelons qu'il y a trois natures possibles d'objet, que l'on peut lire dans les deux bits de poids faible de l'en-tête :

- KIND_RAWDATA (0) : données seulement. En gros, un tel objet représente un tableau de données, par exemple une chaîne de caractères.
- KIND_PTRDATA (1) : pointeurs. En gros, un tel objet représente un constructeur dont les arguments sont les objets cibles des pointeurs.
- KIND_CLOSURE (2) : fermeture, c'est-à-dire que toutes les données sont des pointeurs sauf la première case. En fait, cette première case désigne le pointeur de code, c'est-à-dire l'adresse où se situe le code de la fermeture, la liste de pointeurs dans les cases suivantes représentant l'environnement de la fermeture.

Dans les deux derniers cas, on peut définir la notion de *fils* d'un objet : un objet b est fils d'un objet a si, et seulement si, il existe un pointeur vers b parmi les données de pointeurs de a.

Attention, il se peut également que parmi ces pointeurs, on ait des *pointeurs nuls*, c'est-à-dire qui ne pointent nulle part. En fait, en Cminor, un pointeur nul est représenté par l'entier 32 bits 0. Donc, en Coq, une valeur représentant un pointeur est de la forme Vptr *bloc offset* ou Vint zero. On peut abstraire les pointeurs en utilisant le type option (block×int) et le prédicat suivant :

Inductive val_is_pointer_or_dumb : val → option (block × int) → **Prop** :=
| val_is_pointer : ∀ b i, val_is_pointer_or_dumb (Vptr b i) (Some (b, i))
| val_is_dumb : val_is_pointer_or_dumb (Vint zero) None.

On le définit comme un prédicat car en Coq, on ne peut pas définir de fonction partielle, à moins d'utiliser le type option en cascade.

Ainsi, le pointeur nul est représenté par None, tandis que Vptr *bloc offset* est représenté par Some (*bloc*, *offset*).

Ce prédicat est fonctionnel (les deux cas étant disjoints), et en fait, il est même surjectif, puisqu'on peut facilement construire une fonction, notée val_is_pointer_or_dumb_recip, de type option (block×int)→val telle que pour tout p, on ait val_is_pointer_or_dumb (val_is_pointer_or_dumb_recip p) p. On montre facilement que ce prédicat est injectif.

2.2.1 Liste de pointeurs

Supposons qu'au sein d'une mémoire m on ait un bloc de numéro *heap* où, entre les offsets *until* exclu et i inclus, un pointeur soit stocké tous les 4 octets.

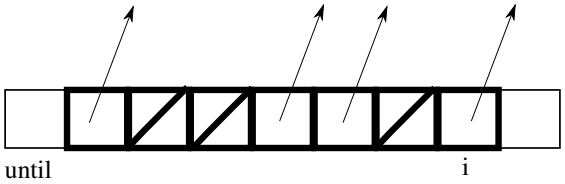

Fig. 17 – Liste de pointeurs

Pour lire cette liste de pointeurs, on dispose d'un prédicat, pointer_list. Attention, on va lire la liste *de droite à gauche* : le premier pointeur de la liste correspond au pointeur le plus à droite dans la mémoire.

Inductive pointer_list (m : mem) ($heap$: block) : int → list (option (block × int)) → **Prop** :=
| pointer_list_end : ∀ i, i = *until* → pointer_list m *heap* i nil
| pointer_list_cont : ∀ i, (i = until → **False**) → ∀ v,
Some v = load Mint32 m *heap* (signed i) → ∀ l, pointer_list m *heap* (sub i four) l →
∀ p, val_is_pointer_or_dumb v p →
pointer_list m *heap* i (p :: l).

Fig. 18 – Liste de pointeurs (pointer_list)

Ce prédicat inductif est défini suivant deux cas disjoints :

- si $i = until$, alors on s'arrête
- sinon, on lit la valeur à l'offset $i-4$, et on concatène le pointeur trouvé à la liste des pointeurs obtenue en continuant la lecture de l'offset $i-4$ jusqu'à *until*

Il est important qu'on lise seulement à l'offset $i-4$ et non à l'offset i. En effet, au départ, on initialise i à un offset qui, en fait, désigne la case mémoire juste à droite de la zone où sont stockés les pointeurs. Cette case mémoire est donc à l'extérieur.

Bien qu'a priori, les pointeurs fils doivent désigner des objets présents dans le tas, la liste obtenue n'est pas une liste d'offsets, mais une liste de pointeurs au sens de l'abstraction définie

ci-avant. En effet, ce prédicat a été conçu pour être réutilisé par exemple pour un GC stop and copy (même si je n'ai finalement pas traité ce type de GC dans ce stage).

Le fait que les deux cas soient disjoints permet d'établir facilement que ce prédicat est fonctionnel : au plus une liste convient pour un jeu de paramètres donné.

De même que pour well_formed_from_to et block_description_from_to, on peut définir un prédicat descriptif, pointer_list_desc, analogue à pointer_list mais en supprimant l'argument liste. Alors, toujours de la même façon que pour la description du tas, si pointer_list_desc est vrai, alors on peut construire dans **Set** une liste de pointeurs qui correspond.

On dispose d'un théorème important concernant l'invariance de cette liste : sous la condition $until \leq i \leq i+3$, si les données entre les offsets *until* et $i+3$ (c'est-à-dire, les valeurs lues 4 octets par 4 octets pour chaque offset entre *until* et i) sont les mêmes entre deux mémoires, alors les listes de pointeurs entre *until* et i sont les mêmes.

On a également un théorème concernant l'invariance par suffixe : si on a deux mémoires m et m' telles que pointer_list est vérifié pour le même *until*, le même i et la même liste l de pointeurs, alors, si pointer_list est vérifié pour la mémoire m, pour le même *until* mais sur un suffixe s de cette liste, alors pointer_list est vérifié pour la mémoire m', pour le même *until* et pour le même s. Ce résultat, assez obscur formulé ainsi, est fort utile dans le cas où on veut parcourir la liste de pointeurs en modifiant la mémoire pour chaque pointeur rencontré. Nous verrons que ce théorème est utilisé dans la preuve de l'algorithme de marquage.

2.2.2 Liste des fils d'un objet

Naïvement, on aurait pu utiliser directement le prédicat pointer_list pour tout objet qui ne soit pas un objet de données (KIND_RAWDATA). Or cela est absurde, car pour une fermeture (KIND_CLOSURE), la première case de données ne contient pas de pointeur vers un objet. Il faut donc distinguer ce cas.

Le prédicat pointer_list_block permet de synthétiser directement les trois natures possibles d'objet.

Ce prédicat, non seulement synthétise les trois cas possibles, mais implique également des hypothèses supplémentaires :

- l'en-tête est lisible (ce qui est vrai dès qu'on considère un objet du tas)
- la nature de l'objet est nécessairement KIND_RAWDATA (0), KIND_PTRDATA (1) ou KIND_CLOSURE (2), ce qui exclut la valeur 3
- si l'objet est une fermeture, alors sa taille est non nulle (ce qui est requis pour pouvoir lire la première case de données, correspondant au pointeur de code)

Les deux dernières conditions n'ont pas été intégrées à la structure du tas, par souci de simplification. En effet, la structure du tas ne régit que les tailles des objets.

On notera bien que la liste des pointeurs est lue via pointer_list avec des arguments *until* différents dans les cas KIND_PTRDATA et KIND_CLOSURE. En effet, dans ce dernier cas, il faut s'arrêter non pas à $b-4$ (offset de l'en-tête, à ne pas lire), mais à b, offset de la première case de données, à ne pas lire non plus.

Comme les trois cas sont disjoints, et que pointer_list est fonctionnelle, pointer_list_block est aussi fonctionnelle.

Comme pour pointer_list, on peut également définir une version descriptive de pointer_list_block, pointer_list_block_desc, en supprimant l'argument liste et en utilisant pointer_list_desc au lieu

```
Inductive pointer_list_block (m : mem) (heap : block) (b : int) : list (option (block × int)) →
Prop :=

| pointer_list_block_rawdata :
∀ v, Some (Vint v) = load Mint32 m heap (signed (sub b four)) →
Kind_header v = KIND_RAWDATA →
pointer_list_block m heap b nil

| pointer_list_block_ptrdata :
∀ v, Some (Vint v) = load Mint32 m heap (signed (sub b four)) →
Kind_header v = KIND_PTRDATA →
∀ l, pointer_list m heap (sub b four) (sub (add b (Size_header v)) four) l →
pointer_list_block m heap b l

| pointer_list_block_closure :
∀ v, Some (Vint v) = load Mint32 m heap (signed (sub b four)) →
Kind_header v = KIND_CLOSURE →
(Size_header v = zero → False) →
∀ l, pointer_list m heap b (sub (add b (Size_header v)) four) l →
pointer_list_block m heap b l.
```

FIG. 19 – Liste des objets fils d'un objet (pointer_list_block)

de pointer_list. Et alors, si pointer_list_block_desc, est vérifiée, alors, en utilisant le théorème analogue pour pointer_list, on peut construire dans **Set** une liste de pointeurs qui correspond.

2.2.3 Bons pointeurs

A priori, les pointeurs que l'on peut trouver dans les listes considérées ici sont quelconques : ils ne pointent pas nécessairement vers des objets du tas. En fait, les propriétés de bonne formation du tas vues en 2.1 ne régissent que les tailles des objets, et non les pointeurs qu'ils contiennent.

Il faut donc définir un autre prédicat pour rajouter cette notion de « bons pointeurs ». Le prédicat good_pointers_from_to exprime que tout pointeur de la liste l a ses objets fils dans l (ou, plus exactement, si $(b,\ o)$ est un couple bloc-offset appartenant à la liste l, alors la liste des ses objets fils est bien définie suivant la nature de l'objet, et tous les pointeurs non nuls de cet objet appartiennent aussi à cette liste).

```
Inductive good_pointers_from_block_list m l : Prop :=
| good_pointers_from_block_list_intro :
(∀ b i, member (b, i) l → pointer_list_block_desc m b i) →
(∀ b i, member (b, i) l → ∀ pl, pointer_list_block m b i pl →
∀ p q, member (Some (p, q)) pl → member (p, q) l) →
good_pointers_from_block_list m l.
```

FIG. 20 – Bons pointeurs à partir d'une liste de pointeurs donnée explicitement (good_pointers_from_block_list)

Il suffit alors d'initialiser l à la liste des objets du tas, à travers block_description_from_to. Attention, alors que l doit être une liste de pointeurs (ou, plus exactement, une liste de couples

bloc-offset), block_description_from_to est une liste d'offsets seulement, cela signifie que tous les pointeurs non nuls doivent pointer au sein du même bloc, celui du tas.

```
Inductive good_pointers_from_to m heap n p : Prop :=
| good_pointers_from_to_intro :
(∀ l, block_description_from_to m heap n p l →
good_pointers_from_block_list (map (fun i ⇒ (heap, i)) l)) →
good_pointers_from_to m heap n p.
```

FIG. 21 – Bons pointeurs à partir de la liste des objets du tas (good_pointers_from_to)

Attention au parenthésage : la liste l n'est pas donnée dans le prédicat good_pointers_from_to, ce qui signifie que good_pointers_from_to n'implique pas nécessairement que le tas est bien formé. Ce prédicat exprime que *si* le tas est bien formé, alors tous les fils des objets du tas sont dans le tas.

2.3 Les pointeurs vers les racines

Un objet racine doit être stocké dans le tas comme tout objet ordinaire. Il n'acquiert le statut de *racine* que par le fait qu'il existe un pointeur vers lui, ce pointeur étant stocké à un endroit spécial.

Dans [MSLL07], les pointeurs vers les racines sont stockés directement dans les registres machine. Ceci est adapté aux gestionnaires de mémoire écrits dans un langage de bas niveau tel que l'assembleur ou même le langage machine. Cependant, cette approche ne peut pas être adoptée dans notre cas, où le gestionnaire de mémoire est écrit dans un langage intermédiaire, Cminor. Ce langage étant destiné à être optimisé, on veut pouvoir bénéficier de ces optimisations pour le gestionnaire de mémoire. Les racines doivent donc être stockées directement en mémoire.

Nous avons choisi de stocker les racines sous la forme d'une liste de listes de racines : les listes de racines sont reliées sous la forme d'une liste chaînée (chaque liste de racines contient un pointeur vers la liste suivante, sauf la dernière liste qui contient un pointeur nul), et chaque liste de racines est constituée d'une case mémoire indiquant le nombre de racines dans la liste, suivi d'autant de cases mémoire qu'il n'en faut, chacune contenant un pointeur vers la racine considérée.

2.3.1 Liste de racines

Une liste de n racines est représentée en mémoire par une suite de $n + 2$ cases au sein d'un même bloc :

- la première case mémoire contient le pointeur vers la liste suivante
- la deuxième case mémoire contient le nombre n de racines, sous la forme d'un entier 32 bits
- chacune des n cases suivantes contient un pointeur vers un objet du tas

Il suffit donc de lire le contenu des n dernières cases.

La structure semble être la même que pour pointer_list. Cependant, ici le principe n'est pas de lire les pointeurs jusqu'à une position donnée, mais en faisant décrémenter le compteur n jusqu'à zéro. On définit alors un prédicat inductif, root_list.

Ce prédicat ne prend pas en compte le nombre effectif de racines. Il suppose ce nombre déjà connu, et permet de lire la liste de façon inductive, comme s'il s'agissait d'une récurrence sur le nombre de racines :

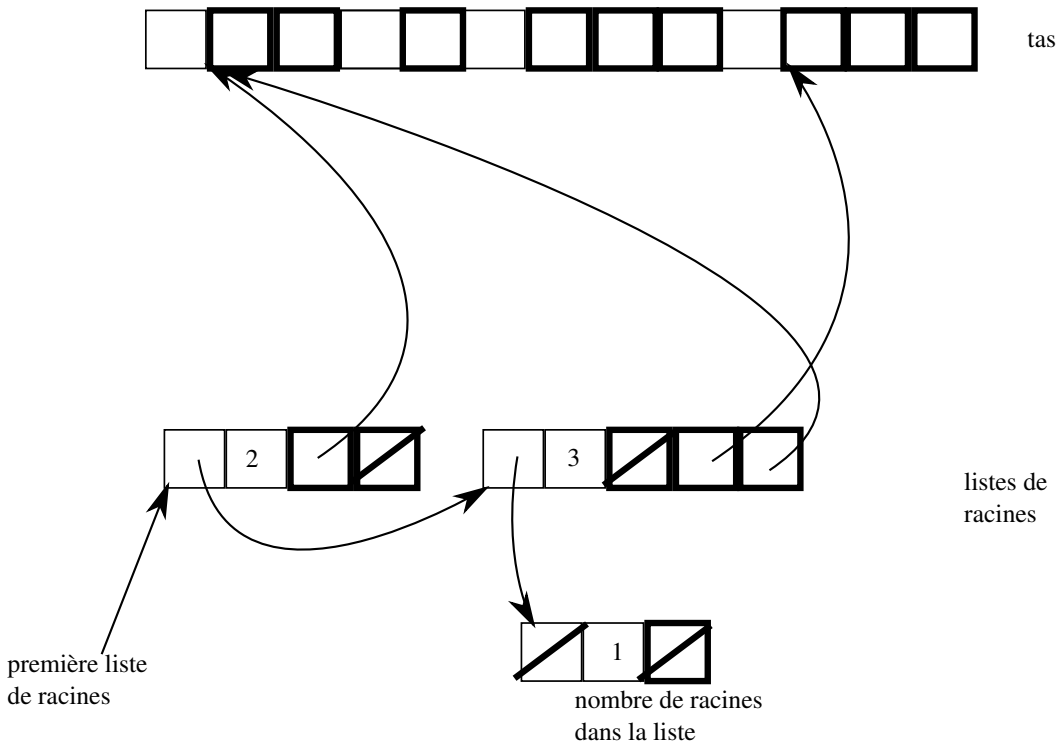

FIG. 22 – Pointeurs vers les objets racine

```
Inductive root_list (m : mem) (r : block) : int → int → list (option (block × int)) → Prop :=
| root_list_zero : ∀ n, n = zero → ∀ i, root_list m r n i nil
| root_list_nonzero : ∀ n, (n = zero → False) →
∀ i, cmp Cle (sub i four) i = true →
∀ bv, Some bv = load Mint32 m r (signed i) →
∀ bv', val_is_pointer_or_dumb bv bv' →
∀ l, root_list m r (sub n one) (add i four) l →
root_list m r n i (bv'::l).
```

FIG. 23 – Liste de racines (root_list)

– Si n est nul, alors la liste de racines est vide
– Sinon, on lit un pointeur à l'offset i, pointeur qu'on ajoute à la liste des n-1 racines suivantes.

Contrairement à pointer_list, la liste est lue ici *de gauche à droite*.

Comme pour les prédicats vus auparavant, ce prédicat est fonctionnel, et on peut définir un prédicat descriptif, en supprimant l'argument liste, que l'on peut reconstruire dans **Set**.

2.3.2 Chaînage

Chaque liste de racines contient en tant que première case mémoire un pointeur vers la liste suivante, ou le pointeur nul dans le cas de la dernière liste de racines.

On connaît à l'avance un pointeur vers la première liste de racines. On peut alors considérer que si ce pointeur est nul, alors la liste de listes de racines est vide.

Par souci de simplicité, on utilise le prédicat all_roots qui prend en argument une *valeur* constituant le pointeur vers la première liste de racines.

Ce prédicat inductif est défini suivant deux cas disjoints (il est donc fonctionnel) :

```
Inductive all_roots (m : mem) : val → list (list (option (block × int))) → Prop :=
| all_roots_zero : all_roots m (Vint zero) nil
| all_roots_nonzero : ∀ r d,
cmp Cle d (add d four) = true →
∀ n, Some (Vint n) = load Mint32 m r (signed (add d four)) →
∀ l, root_list m r n (add (add d four) four) l →
∀ v', Some v' = load Mint32 m r (signed d) →
∀ l', all_roots v' l' →
all_roots (Vptr r d) (l :: l').
```

FIG. 24 – Liste des listes de racines (all_roots)

- le cas du pointeur nul : la liste de listes de pointeurs est vide
- le cas d'un pointeur vers la liste de racines : dans ce cas, on lit alors la deuxième case mémoire, qui contient le nombre n de racines ; on utilise alors root_list pour lire cette liste de n racines à partir de la troisième case mémoire ; puis on lit la première case mémoire qui contient un pointeur vers la liste suivante, pointeur à partir duquel on lit à nouveau la liste des listes de racines restantes.

La liste l obtenue est alors une liste de listes de racines. Pour exprimer qu'un pointeur p appartient à cette liste de listes, il faut écrire member p (flatten l). (Voir la librairie List_addons en annexe.)

On notera que, bien que chaque liste de racines occupe un seul bloc, on peut avoir plusieurs listes de racines dans un même bloc.

Comme pour tous les autres prédicats vus jusqu'ici, on peut définir une version descriptive all_roots_desc de all_roots en supprimant l'argument liste que l'on peut retrouver dans **Set**.

2.3.3 Lieux de stockage des pointeurs vers les racines. Invariance des listes de racines.

La représentation en mémoire de la liste de listes de racines n'est pas évidente : elle peut s'étendre sur plusieurs blocs. Pour pouvoir établir des résultats d'invariance (c'est-à-dire exprimer que la liste des listes de racines est inchangée entre deux mémoires), il faut alors exprimer, par exemple, que les valeurs lues aux positions où on lit les pointeurs vers les racines sont inchangées. Il faut donc décrire ces positions.

L'idée est de reprendre les prédicats root_list et all_roots en remplaçant l'argument liste de (liste de) pointeurs par un argument liste de (liste de) positions. Une position est représentée par une valeur COQ de type block×int. On obtient alors respectivement les prédicats root_list_position_list et root_position_list.

Dans le cas de root_list_position_list, on ne considère que la liste des positions contenant effectivement les racines (i.e. la liste des positions nécessaires à la lecture de root_list). On ne s'occupe pas des deux premières cases mémoire. Celles-ci seront rajoutées par la suite avec le prédicat root_position_list. Aussi remarque-t-on que root_list_position_list ne dépend pas de la mémoire.

Ces deux prédicats inductifs sont fonctionnels, et les listes de positions obtenues peuvent être calculées dans **Set** dès lors qu'on dispose des hypothèses respectives root_list_desc et all_roots_desc.

On peut alors établir les résultats d'invariance suivants :

```
Inductive root_list_position_list : block → int → int → list (block × int) → Prop :=
| root_list_pl_zero : ∀ n, n = zero → ∀ r i, root_list_position_list r n i nil
| root_list_pl_nonzero : ∀ n, (n = zero → False) →
∀ i, cmp Cle (sub i four) i = true →
∀ r l, root_list_position_list r (sub n one) (add i four) l →
root_list_position_list r n i ((r, i) : :l).

Inductive root_position_list : val → list (list (block × int)) → Prop :=
| root_pl_zero : root_position_list (Vint zero) nil
| root_pl_nonzero : ∀ r i,
cmp Cle i (add i four) = true →
∀ n, Some (Vint n) = load Mint32 m r (signed (add i four)) →
∀ l, root_list_position_list r n (add (add i four) four) l →
∀ v', Some v' = load Mint32 m r (signed i) →
∀ l', root_position_list v' l' →
root_position_list (Vptr r i) (((r, i) : : (r, add i four) : : l) : : l').
```

FIG. 25 – Positions où sont stockés les pointeurs vers les racines (root_list_position_list, root_position_list)

- Si les lectures sont les mêmes aux positions déclarées par root_list_position_list, alors les listes de racines données par root_list sont les mêmes.
- Si on a deux mémoires m et m' telles que les lectures aux positions déclarées par root_position_list pour m sont les mêmes dans les deux mémoires, alors les listes de listes de racines données par all_roots sont les mêmes dans les deux mémoires, ainsi que les positions déclarées par root_position_list.

Cependant, on dispose également de résultats réciproques :

- Si les listes de racines données par root_list sont les mêmes, alors les lectures sont les mêmes aux positions déclarées par root_list_position_list.
- Si deux mémoires m et m' sont telles que les listes de listes de racines données par all_roots sont les mêmes dans les deux mémoires, ainsi que les positions déclarées par root_position_list, alors les lectures à ces positions sont les mêmes.

2.4 Chemin dans la mémoire

Grâce aux structures de tas, d'objets fils et de racines, on peut modéliser la mémoire comme une forêt dont les racines sont justement les objets racine, et les noeuds sont les objets eux-mêmes. On peut donc définir la notion de chemin dans cette forêt.

Cette notion est pertinente au niveau de la traduction des langages de programmation de haut niveau, car :

- savoir si un objet est utilisé revient à exhiber un chemin vers cet objet
- si on modifie les chemins (c'est-à-dire, si on exécute un programme bas niveau, tel qu'un GC, et que les objets utilisés sont les mêmes après qu'avant l'exécution mais suivant des chemins différents), alors les objets structurés au haut niveau ne sont plus les mêmes.

Par exemple, en OCaml, supposons qu'on ait un couple k de deux références, notons $k = (r_1, r_2)$, avec r2 contenant une référence r3 contenant la valeur 18, et r1 contenant, disons, la valeur 42.

En mémoire, le couple est représenté par un objet à deux fils, l'un vers la référence r1, l'autre vers la référence r2. Chaque référence est représentée par un objet à un fils.

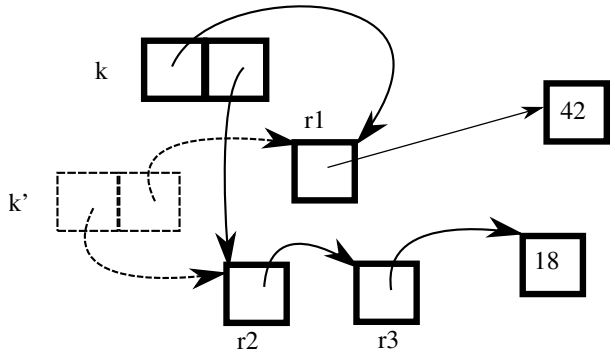

FIG. 26 – Objets accessibles par des chemins différents depuis k et k'

Supposons que le couple k soit une racine. Alors, la référence r3 est accessible depuis cette racine, en suivant le chemin « deuxième fils, puis premier fils ». Mais si un GC mal écrit passe et intervertit les pointeurs au niveau du couple k, alors r3 restera accessible mais par le chemin différent « premier fils, puis premier fils », et la mémoire ne représentera plus k mais un autre couple, (r_2, r_1).

Cette notion constitue une grande partie de l'interface entre la preuve du GC et celle du compilateur certifié de MiniML vers Cminor, mais qui n'a pas été traitée dans le cadre de ce stage.

2.4.1 Définition

Un *chemin* est la donnée de :

- un entier naturel, numéro de la liste de racines où chercher le pointeur vers la racine
- un entier naturel, numéro du pointeur vers la racine au sein de la liste de racines
- une liste d'entiers naturels, indiquant les numéros des objets fils à visiter successivement jusqu'à épuisement de la liste (la *branche* à suivre)

```
Record path : Set := mkPath {
root_list_number : nat;
root_number : nat;
pointer_chain : list nat
}.
```

FIG. 27 – Chemin dans la mémoire (path)

Un chemin est donné indépendamment de la mémoire. On ne connaît donc pas a priori quel est l'objet désigné par ce chemin. Déterminer cet objet, c'est *réaliser* (ou *interpréter*) le chemin. Cependant, il y a deux niveaux de réalisation :

- la réalisation *concrète* à partir de la mémoire elle-même
- la réalisation *abstraite* à partir d'une représentation abstraite de la mémoire sous la forme d'une forêt

2.4.2 Réalisation concrète

La réalisation concrète du chemin est définie directement à travers des prédicats qui lisent les valeurs correspondantes, indépendamment de la structure de tas de la mémoire. Cependant, la structure de listes de listes de racines reste requise.

On commence par chercher la liste de racines. Disposant d'un entier naturel root_list_number indiquant le numéro de cette liste, il suffit en fait de suivre une chaîne de pointeurs en faisant décrémenter root_list_number à chaque pointeur suivi jusqu'à zéro. On obtient alors le prédicat find_root_list.

```
Inductive find_root_list : nat → block → int → block → int → Prop :=
| find_root_list_O : ∀ p q, find_root_list O p q p q
| find_root_list_S : ∀ b i c j,
Some (Vptr c j) = load Mint32 m b (signed i) → ∀ n d k, find_root_list n c j d k →
find_root_list (S n) b i d k.
```

FIG. 28 – Recherche d'une liste de racines (find_root_list)

Ce prédicat calcule le bloc et l'offset de la première case de la liste de racines voulue. Notons toutefois qu'il requiert le bloc et l'offset de la première case de la première liste de racines : aucun chemin ne peut donc être réalisé concrètement si la liste de listes de racines est vide (i.e. si le pointeur vers la première liste de racines est nul).

Une fois la liste de racines trouvée, il faut trouver la racine. Disposant d'un entier naturel root_number, mais aussi d'un entier 32 bits n indiquant le nombre de racines dans la liste, il faut se déplacer vers la droite dans la liste de racines en faisant décrémenter à la fois l'entier naturel root_number, et l'entier 32 bits n, jusqu'à ce que root_number tombe à zéro mais tout en maintenant n non nul à chaque fois. Ceci permet d'éviter d'avoir à effectuer des conversions entre entiers naturels et entiers 32 bits. On obtient alors le prédicat find_root.

```
Inductive find_root : nat → int → block → int → option (block × int) → Prop :=
| find_root_number_O : ∀ n, (n = zero → False) →
∀ r d pv, Some pv = load Mint32 m r (signed d) →
∀ v, val_is_pointer_or_dumb pv v →
find_root O n r d v
| find_root_number_S : ∀ n, (n = zero → False) →
∀ k r d pv, find_root k (sub n one) r (add d four) pv →
find_root (S k) n r d pv.
```

FIG. 29 – Recherche d'une racine dans une liste de racines (find_root)

Une fois la racine trouvée, il faut en partir en suivant la branche indiquée dans le chemin sous la forme de la liste d'entiers naturels pointer_chain.

Pour chaque objet considéré, dont on notera $o - 4$ l'offset de l'en-tête, on veut connaître le p-ième fils, où p est un entier naturel. Pour un objet de pointeurs comme pour un objet de fermeture, on considère que le premier fils (de numéro zéro) est le pointeur le plus à droite (c'est-à-dire, le plus éloigné de l'en-tête). On lit donc de droite à gauche, en décrémentant p à chaque fois jusqu'à tomber à zéro, mais en prenant garde à ne pas atteindre la limite de lecture, c'est-à-dire $o-4$ dans le cas d'un objet de pointeurs, o dans le cas d'une fermeture. On obtient donc deux prédicats : find_pointer_from consiste à chercher le pointeur dans la liste jusqu'à une

certaine limite de lecture, et find_pointer utilise ce prédicat en initialisant la limite de lecture à $o-4$ ou à o en fonction de la nature de l'objet, en lisant l'en-tête.

```
Inductive find_pointer_from : nat → int → option (block × int) → Prop :=
| find_pointer_O : ∀ i, (i = I → False) →
∀ pv, Some pv = load Mint32 m b (signed i) →
∀ opv, val_is_pointer_or_dumb pv opv →
find_pointer_from O i opv
| find_pointer_S : ∀ i, (i = I → False) →
∀ n pv, find_pointer_from n (sub i four) pv →
find_pointer_from (S n) i pv.
```

FIG. 30 – Recherche d'un pointeur dans une liste de pointeurs (find_pointer_from)

```
Inductive find_pointer : nat → int → option (block × int) → Prop :=
| find_pointer_ptrdata : ∀ i v, Some (Vint v) = load Mint32 m b (signed (sub i four)) →
Kind_header v = KIND_PTRDATA →
∀ n pw, find_pointer_from (sub i four) n (sub (add i (Size_header v)) four) pw →
find_pointer n i pw
| find_pointer_closure : ∀ i v, Some (Vint v) = load Mint32 m b (signed (sub i four)) →
Kind_header v = KIND_CLOSURE →
∀ n pw, find_pointer_from i n (sub (add i (Size_header v)) four) pw →
find_pointer n i pw.
```

FIG. 31 – Recherche d'un objet fils d'un objet du tas (find_pointer)

Le prédicat follow_pointer_chain permet alors de suivre la branche, en utilisant le prédicat find_pointer pour chaque objet visité.

```
Inductive follow_pointer_chain : option (block × int) → list nat → option (block × int) → Prop :=
| follow_pointer_chain_nil : ∀ bi, follow_pointer_chain bi nil bi
| follow_pointer_chain_cons : ∀ b n i cj, find_pointer b n i cj →
∀ l dk, follow_pointer_chain cj l dk → follow_pointer_chain (Some (b, i)) (n : :l) dk.
```

FIG. 32 – Suivi d'une branche (follow_pointer_chain)

Tous ces prédicats sont synthétisés dans realize_path.

Bizarrement, ces prédicats semblent constructifs, à tel point qu'ils auraient même pu être définis sous la forme de fonctions à valeurs dans des types option, par récurrence Fixpoint sur les entiers naturels indiquant le numéro de la liste des racines, le numéro de la racine, etc. ou sur la liste des numéros d'objets fils. Cependant, le type-checker de Coq 8.1 boucle (sans augmenter la mémoire utilisée) au moment de la définition de telles fonctions. Il faut alors passer par des structures de boucle plus générales (avec la librairie Loop présentée en annexe), en dépit du fait que ces boucles sont structurelles. Des exemples de telles définitions constructives sont présentes dans le module Constrpath ; elles n'ont pas été utilisées dans ce stage, car pour le raisonnement il est beaucoup plus facile d'utiliser les prédicats.

On dispose de théorèmes d'invariance de la réalisation concrète (c'est-à-dire que les réalisations concrètes de tout chemin sont les mêmes entre les deux mémoires m et m' considérées), sous les conditions que :

```
Inductive realize_path :
val →
(path) →
(option (block × int)) →
Prop :=
| realize_path_intro : ∀ r d p qv r0 d0, find_root_list p.(root_list_number) r d r0 d0 →
∀ n, Some (Vint n) = load Mint32 m r0 (signed (add d0 four)) →
∀ b0u0, find_root p.(root_number) n r0 (add (add d0 four) four) b0u0 →
follow_pointer_chain b0u0 p.(pointer_chain) qv →
realize_path (Vptr r d) p qv.
```

FIG. 33 – Réalisation concrète d'un chemin (realize_path)

- les structures de tas soient les mêmes pour m et pour m'
- m admette les bons pointeurs (tous les objets fils d'un objet du tas de m sont dans le tas de m)
- les natures des objets du tas, ainsi que leurs données, soient les mêmes pour m et pour m'
- les objets racine de m soient dans le tas de m
- les valeurs aux positions des pointeurs vers les racines soient les mêmes pour m et pour m'

Mais on dispose également de théorèmes d'invariance de la réalisation concrète *par chemin*, c'est-à-dire en supposant que seuls les objets accessibles par un chemin (c'est-à-dire, les objets qui sont réalisations concrètes d'un chemin) sont inchangés. Ces théorèmes sont donc indépendants de la structure du tas. Supposons qu'on ait deux mémoires m et m'. Alors, si les conditions suivantes sont réunies :

- les valeurs aux positions des pointeurs vers les racines sont les mêmes pour m et pour m'
- pour tout objet accessible dans m, si $o-4$ est l'offset de son en-tête, et s sa taille (lue dans l'en-tête), alors on a $o-4 \leq o+s-1$ (i.e. les objets accessibles sont constitués de cases mémoire contiguës)
- tout objet de fermeture accessible dans m, a une taille non nulle
- les cases de données et l'en-tête des objets accessibles dans m ont la même valeur dans m que dans m'

Alors, un chemin qui se réalise concrètement dans m se réalise aussi concrètement dans m' et désigne le même objet.

La réciproque est vraie si de plus tout objet accessible dans m admet un en-tête.

Ces deux théorèmes peuvent être utilisés dans les algorithmes de sweep et d'allocation, où seuls les blocs libres (c'est-à-dire non accessibles) sont modifiés, à cause respectivement de la coalescence, et de l'allocation.

2.4.3 Représentation abstraite de la mémoire

Comme on l'a vu, les prédicats de réalisation concrète peuvent faire fi de la notion de structure du tas. Cela peut être assez dangereux notamment pour l'allocation : on pourrait avoir des mémoires à tas lâche, c'est-à-dire où entre deux objets il subsiste des cases mémoire ne décrivant pas un objet du tas. On ne veut pas considérer cette éventualité, qui pourrait compliquer les preuves.

Au contraire, on veut pouvoir représenter la mémoire sous la forme abstraite d'une forêt dans laquelle on décrirait les chemins. Ainsi, il suffirait alors, pour deux mémoires m et m', d'être

représentées par la même forêt abstraite pour que les chemins dans ces deux mémoires soient les mêmes (c'est-à-dire, un chemin p est réalisé concrètement dans m vers l'objet o si et seulement si p est réalisé concrètement dans m' vers o).

On considère donc trois étapes :

- abstraction de la mémoire par une forêt
- définition d'une réalisation abstraite d'un chemin dans la forêt
- coïncidence entre la réalisation abstraite et la réalisation concrète dans le cas où une mémoire peut être abstraite par une forêt

Un état mémoire abstrait est représenté par deux types record. Les champs de ces enregistrements correspondent respectivement aux données suivantes :

- pour memory_param (paramètres initiaux de la mémoire) :
  - le bloc mémoire du tas
  - l'offset de début de tas
  - l'offset de fin de tas (qui désigne la première case mémoire *après* le tas)
  - un pointeur vers la première liste des racines (ou le pointeur nul)
- pour used_abstraction_param (structure du tas et des racines)
  - la liste des offsets des objets du tas
  - les tailles des objets du tas
  - les natures des objets du tas
  - un booléen indiquant si les deux bits de poids fort des en-têtes de tous les objets sont égaux à zéro (ceci doit être vrai en dehors des étapes de ramasse-miettes)
  - les données des objets de nature KIND_RAWDATA
  - la liste des objets fils de chaque objet
  - la liste des listes des positions où sont stockés les pointeurs vers les racines
  - la liste des listes des racines

```
Record memory_param : Set := make_memory_param {
memory_heap : block ;
memory_hs : int ;
memory_he : int ;
memory_rd : val
}.
```

FIG. 34 – Paramètres initiaux de la mémoire (memory_param)

```
Record used_abstraction_param : Set := make_used_abstraction_param {
used_blocklist : list int ;
used_sizes : int → int ;
used_kinds : int → int ;
used_extra_zero : bool ;
used_rawdata : int → val ;
used_children : option (block × int) → list (option (block × int)) ;
used_rootpos : list (list (block × int)) ;
used_roots : list (list (option (block × int)))
}.
```

FIG. 35 – Paramètres de la structure abstraite de tas et racines (used_abstraction_param)

La relation d'abstraction est donnée par le type enregistrement used_abstraction dans **Prop**.

```
Record used_abstraction (m : mem) (mp : memory_param) (p : used_abstraction_param) :
Prop := used_abstraction_intro {

used_heap_hyp : block_description_from_to m (memory_heap mp) (memory_hs mp) (memory_he
mp) (used_blocklist p);
used_sizes_hyp : ∀ i, member i (used_blocklist p) → ∀ v, Some (Vint v) = load Mint32 m
(memory_heap mp) (signed (sub i four)) →
Size_header v = used_sizes p i;
used_kinds_hyp : ∀ i, member i (used_blocklist p) → ∀ v, Some (Vint v) = load Mint32 m
(memory_heap mp) (signed (sub i four)) →
Kind_header v = used_kinds p i;

used_extra_zero_hyp : used_extra_zero p = true → ∀ i, member i (used_blocklist p) → ∀
v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub i four)) →
Extra_header v = zero;

used_rawdata_hyp : ∀ i, member i (used_blocklist p) → used_kinds p i = KIND_RAWDATA →
(used_sizes p i = zero → False) → ∀ x,
cmp Cle i x = true → cmp Cle x (sub (add i (used_sizes p i)) four) = true →
modulo_four x i → load Mint32 m (memory_heap mp) (signed x) = Some (used_rawdata p x);

used_children_hyp : ∀ b, member b (used_blocklist p) → pointer_list_block m (memory_heap mp)
b (used_children p (Some (memory_heap mp, b)));

used_children_are_blocks : ∀ b, member b (used_blocklist p) → ∀ cb co, member (Some
(cb, co)) (used_children p (Some (memory_heap mp, b))) →
(cb = memory_heap mp ∧ member co (used_blocklist p));

used_children_none : used_children p None = nil;

used_rootpos_hyp : root_position_list m (memory_rd mp) (used_rootpos p);

used_roots_hyp : all_roots m (memory_rd mp) (used_roots p);

used_roots_are_blocks : ∀ r' d', member (Some (r', d')) (flatten (used_roots p)) → (r' =
memory_heap mp ∧ member d' (used_blocklist p));

used_rootpos_not_in_heap : ∀ b i, member (b, i) (flatten (used_rootpos p)) → b ≠ (memory_heap
mp)
}.
```

FIG. 36 – Relation d'abstraction du tas et des racines (used_abstraction)

Les assertions de la relation d'abstraction sont les suivantes :

- le tas est bien formé et produit la liste d'offsets attendue
- la fonction used_sizes calcule les natures des objets du tas lues dans les en-têtes
- la fonction used_kinds calcule les natures des objets du tas lues dans les en-têtes

- si le booléen used_extra_zero est vrai, alors les deux bits de poids fort des en-têtes de tous les objets sont égaux à zéro. Cette propriété est utile si on modifie une en-tête d'un bloc, pour montrer que les autres en-têtes sont inchangés, mais on n'a besoin ici que d'un seul sens de l'implication ; mettre ce booléen à false permet d'ignorer l'hypothèse, et donc permet au GC d'utiliser ces deux bits de poids fort comme bon lui semble ; dans les preuves, il faut alors gérer les en-têtes plus finement, à un niveau plus concret.
- la fonction used_rawdata calcule les données des objets de données[7]
- la fonction used_children calcule la liste des objets fils de chaque objet du tas (et par conséquent, comme on l'a vu en définissant pointer_list_block, les objets du tas sont nécessairement des données, des listes de pointeurs ou des fermetures, et les fermetures sont nécessairement des objets de taille non nulle)
- les objets fils d'un objet du tas sont des objets du tas (par conséquent, la mémoire a les « bons pointeurs »)
- le pointeur nul n'a pas d'objets fils
- la fonction used_rootpos calcule la liste des listes des positions où sont stockés les pointeurs vers les racines
- la fonction used_roots calcule la liste des listes des objets racine
- les racines sont des objets du tas (plus exactement, un pointeur vers une racine soit est nul, soit pointe vers un objet du tas)
- les pointeurs vers les racines ne sont pas stockés dans le bloc mémoire du tas (cette propriété est nécessaire notamment pour montrer que si le tas est modifié, alors les pointeurs vers les racines sont inchangés)

Beaucoup de prédicats qu'il semble difficile de devoir prouver d'un coup pour une mémoire donnée à l'avance. Cependant, on peut prouver que, étant donné une mémoire m, si toutes les conditions suivantes sont réunies :

- le tas est bien formé
- la mémoire m a les bons pointeurs (c'est-à-dire que, tous les objets du tas ont leurs objets fils dans le tas)
- la structure des pointeurs vers les racines est bien formée
- les racines sont des objets du tas
- les positions où sont stockées les pointeurs vers les racines ne sont pas dans le tas

Alors, on peut construire un état abstrait pour une telle mémoire m. Bien sûr, ces conditions sont aussi impliquées par la relation d'abstraction. Elles sont donc nécessaires et suffisantes. Mais de plus, l'état abstrait peut être construit dans Set, grâce aux analogues dans Set pour les prédicats block_description_from_to, pointer_list_block et all_roots.

2.4.4 Réalisation abstraite

La notion de réalisation abstraite d'un chemin est définie uniquement à partir de la donnée d'un terme *up* de type used_abstraction, donc indépendamment de la mémoire. Elle est définie à base du prédicat find_in_list de la librairie List_addons (décrite en annexe), qui permet de trouver un élément d'une liste de façon indexée comme s'il s'agissait d'un tableau, la tête ayant l'index 0. Ce prédicat est équivalent à member (apparaître dans la liste) à ceci près qu'il contient l'information supplémentaire de la position de l'élément dans la liste. Ainsi, il n'est plus nécessaire de lire directement dans la mémoire pour suivre le chemin : l'état abstrait suffit.

[7] Il faudrait également ajouter qu'elle calcule aussi la première case mémoire des objets de fermeture. Cet ajout est aisé mais requiert de revérifier toutes les preuves du GC.

Alors, dans le cadre de la réalisation abstraite, trouver le n-ième objet fils d'un objet o du tas revient à trouver le n-ième objet de la liste used_children up de ses fils. À partir de là, on peut alors définir comment suivre une branche (c'est-à-dire, trouver successivement les objets fils à partir des indices fournis dans une liste), grâce au prédicat suivant :

```
Inductive used_follow p : option (block × int) → list nat → option (block × int) → Prop :=
| used_follow_nil : ∀ a, used_follow p a nil a
| used_follow_cons : ∀ a n b, find_in_list (used_children p a) n b → ∀ l c, used_follow p b l c →
used_follow p a (n : :l) c
.
```

FIG. 37 – Suivi d'une branche d'un chemin dans le tas abstrait (used_follow)

Étant donné un état mémoire abstrait up (de type used_abstraction_param), on dit qu'un chemin p est *réalisé abstraitement* dans up vers l'objet o si et seulement si toutes les conditions suivantes sont réunies :

- l'élément numéro root_list_number p de la liste des listes des racines used_root_list up est une liste rl
- l'élément numéro root_number p de la liste des racines rl trouvée auparavant, est un pointeur ptr (de type option (block×int))
- la branche est suivie abstraitement de ptr à l'objet o

```
Inductive used_realize_path (up : used_abstraction_param) (p : path) (bi : option (block × int)) :
Prop :=
| used_realize_path_intro : ∀ rl,
find_in_list (used_roots up) (root_list_number p) rl →
∀ r, find_in_list rl (root_number p) r →
used_follow up r (pointer_chain p) bi →
used_realize_path up p bi.
```

FIG. 38 – Réalisation abstraite d'un chemin (used_realize_path)

On peut alors montrer que, si l'état mémoire abstrait est associé à une mémoire concrète par la relation d'abstraction used_abstraction, alors pour tout chemin, sa réalisation concrète dans l'état mémoire abstrait et sa réalisation concrète dans la mémoire associée aboutissent au même objet (ou aucune des deux n'aboutit). Cette preuve se fait en montrant d'abord, pour chacun des prédicats find_root_list, find_root, find_pointer et follow_pointer_chain de la réalisation concrète, que ces prédicats de recherche équivalent chacun à la recherche d'un élément dans la liste correspondante (liste des listes de racines, liste de racines, liste des objets fils).

Une conséquence de ce théorème est que si deux mémoires sont associées à un même état mémoire abstrait, alors elles ont les mêmes réalisations concrètes pour tous les chemins.

C'est la raison pour laquelle c'est cet état mémoire abstrait qui va servir notamment pour l'étape de marquage, où la structure du tas n'est pas modifiée : on va montrer que tout au long de l'étape de marquage, l'état mémoire abstrait reste constant, donc les réalisations, tant abstraites que concrètes, de tous les chemins resteront inchangées. Et alors, il suffit de montrer uniquement la relation d'abstraction, et toutes les preuves pourront faire fi des chemins : pour le marquage, on n'utilisera plus qu'une relation d'accessibilité faible, sans donnée du chemin.

Notons toutefois que la réalisation abstraite ne dépend en fait que de la liste des listes des racines, et de la fonction qui calcule les objets fils. Le théorème ci-dessus peut donc être affiné

pour s'étendre aux mémoires qui s'abstraient vers des états ayant seulement les mêmes racines et les mêmes objets fils (indépendamment du tas, par exemple). Il pourrait alors être utilisé pour l'allocation et le sweep.

En exhibant ainsi deux façons distinctes de réaliser un chemin, on montre ainsi qu'il est utile de dissocier la notion de chemin de celle de son interprétation : ainsi, la porte est ouverte pour définir encore une autre interprétation d'un chemin dans le cas d'un GC stop and copy, où les chemins se réalisent à cheval sur deux tas distincts, l'ancien tas en cours de copie et le nouveau tas en cours de construction, avec des passages de l'un à l'autre, la notion de chemin, quant à elle, restant la même.

3 Le marquage

Dans ce stage nous nous sommes proposés de prouver la correction d'une implémentation en Cminor d'un GC mark and sweep.

L'algorithme de marquage considéré est un algorithme séquentiel (c'est-à-dire non récursif) utilisant un *cache partiel* : les objets en attente d'être marqués utilisés sont à la fois coloriés en gris et ajoutés dans un cache (s'il n'est pas déjà plein). Une variable booléenne est utilisée pour savoir s'il peut exister des objets gris en dehors du cache. Cette situation se produit si le programme a essayé d'ajouter un élément au cache alors qu'il était déjà plein.

1. Marquage en gris et mise en cache des racines.
2. Si le cache est non vide, on retire un élément du cache.
3. Si le cache est vide, mais que tous les objets gris sont nécessairement dans le cache, alors le marquage termine.
4. Sinon, on cherche un objet gris directement dans le tas. S'il n'y en a pas, alors le marquage termine.
5. L'objet gris considéré est marqué en noir.
6. Chacun de ses enfants est marqué en gris avec la fonction mark_block.
7. On retourne à l'étape 2.

Une implémentation en Cminor a été proposée par Xavier Leroy. J'en ai proposé une simplification pour la preuve, sans incidence sur les performances algorithmiques. Puis j'ai écrit en Coq les algorithmes correspondants manipulant la mémoire directement, afin de m'affranchir des subtilités de la sémantique du langage Cminor. Je montre dans un premier temps comment relier cette implémentation à ces algorithmes, en prouvant que les états mémoire en sortie sont les mêmes dans les deux cas.

Pour ce qui est de la preuve du marquage proprement dite, la structure de tas de la mémoire n'est pas modifiée. Alors on peut considérer des algorithmes abstraits de marquage qui ne considèrent plus la mémoire elle-même, mais uniquement les objets, à qui ils attribuent une couleur, et prouver que ces algorithmes colorient en noir tous les objets utilisés et seulement eux.

Je prouve alors que les algorithmes de marquage concret conservent la structure de tas et colorient les objets de la même façon que les algorithmes de marquage abstrait. Ainsi, à la fin du marquage, les objets marqués en noir sont tous les objets utilisés et seulement eux.

3.1 Description de l'implémentation

3.1.1 Les programmes en Cminor

Xavier Leroy a proposé une implémentation de l'algorithme de marquage en Cminor. Elle est composée de trois procédures :

- mark_block : marquage d'un bloc en gris
- find_first_gray_block : recherche d'un élément gris dans le tas
- gc_mark : le marquage lui-même

L'algorithme repose sur une fonction de marquage en gris, mark_block. Celle-ci prend en paramètre un objet blanc, non encore marqué. S'il s'agit d'un objet de données, alors il est marqué en noir. Sinon, il est colorié en gris et ajouté au cache, géré comme une pile. Cependant, le cache a une taille limitée. Il se peut alors que le cache soit vidé mais qu'il reste encore des objets gris dans le tas. Dans ce cas, il faut chercher un objet gris directement dans le tas.

La couleur des objets est stockée dans leur en-tête.

```
var "heap_start"[4]
var "heap_end"[4]
#define GRAY_CACHE_SIZE 65536
var "gray_cache"[GRAY_CACHE_SIZE]
var "gray_cache_ptr"[4]
var "gray_cache_overflow"[4]
#define KIND_RAWDATA 0
#define KIND_PTRDATA 1
#define KIND_CLOSURE 2
#define Kind_header(h) ((h) & 3)
#define COLOR_WHITE 0
#define COLOR_GRAY 4
#define COLOR_BLACK 0xC
#define Color_header(h) ((h) & 0xC)
#define Size_header(h) (((h) >>u 2) & 0xFFFFFFFC)
```

Les macros `#define` sont traitées par un préprocesseur, qui effectue les substitutions désirées directement dans le code.

FIG. 39 – Algorithme de marquage d'origine : variables globales et macros

```
"mark_block"(b): int -> void
{
  stack 0;
  var header, cache;
  if (b == 0) return;
  header = int32[b - 4];
  if (Color_header(header) != COLOR_WHITE) return;
  if (Kind_header(header) == KIND_RAWDATA) {
    /* Set it to black now, as there are no pointers within */
    int32[b - 4] = header | COLOR_BLACK;
  } else {
    int32[b - 4] = header | COLOR_GRAY;
    /* Is there room in the gray_cache? */
    cache = int32["gray_cache_ptr"];
    if (cache == "gray_cache" + GRAY_CACHE_SIZE) {
      int32["gray_cache_overflow"] = 1;
    } else {
      int32[cache] = b;
      int32["gray_cache_ptr"] = cache + 4;
    }
  }
}
```

FIG. 40 – Algorithme de marquage d'origine : coloration d'un élément en gris

```
"find_first_gray_block"(): int
{
  stack 0;
  var p, lastp, header;
  p = int32["heap_start"];
  lastp = int32["heap_end"];
  loop {
    if (p >= lastp) return 0;
    header = int32[p];
    if (Color_header(header) == COLOR_GRAY) return p + 4;
    p = p + 4 + Size_header(header);
  }
}
```

FIG. 41 – Algorithme de marquage d'origine : recherche d'un élément gris du tas

```
"gc_mark"(root) : int -> void
{
  var numroots, p, cache, b, header, firstfield, n;
  int32["gray_cache_ptr"] = "gray_cache";
  int32["gray_cache_overflow"] = 0;
  {{ loop {
    if (root == 0) exit;
    numroots = int32[root + 4];
    p = root + 8;
    {{ loop {
      if (numroots == 0) exit;
      "mark_block"(int32[p]) : int -> void;
      p = p + 4;
      numroots = numroots - 1;
    } }}
    root = int32[root];
  } }}
  {{ loop {
    /* Find next gray object to work on */
    cache = int32["gray_cache_ptr"];
    if (cache > "gray_cache") {
      cache = cache - 4;
      b = int32[cache];
      int32["gray_cache_ptr"] = cache;
    } else {
      if (int32["gray_cache_overflow"] == 0) exit;
      b = "find_first_gray_block"() : int;
      if (b == 0) exit;
    }
    /* b is a gray object of kind PTRDATA or CLOSURE */
    header = int32[b - 4];
    int32[b - 4] = header | COLOR_BLACK;
    /* Call mark_block on all (pointer) fields of b.
       Process fields from last to first since this results
       in better gray_cache utilization in case of right-oriented
       data structures such as lists */
    firstfield = (Kind_header(header) == KIND_CLOSURE) << 2;
    n = Size_header(header);
    {{ loop {
      if (n == firstfield) exit;
      n = n - 4;
      "mark_block"(int32[b + n]) : int -> void;
    } }}
  } }}
}
```

FIG. 42 – Algorithme de marquage d'origine

Idéalement, le cache contient tous les objets gris. Mais il a une taille fixe, GRAY_CACHE_SIZE (sa taille en octets, multiple de 4). Il se peut donc qu'il ne puisse pas contenir tous les objets gris. La variable gray_cache_overflow indique si le programme a essayé au moins une fois d'ajouter un objet à un cache déjà plein. Au moment de la mise en cache d'un objet, si le cache est déjà plein, alors cette variable est mise à 1 (vrai). Elle n'est mise à zéro qu'au début de l'algorithme de marquage, jamais pendant. Alors, la valeur de cette variable est 0 si tous les objets sont nécessairement dans le cache, et 1 s'il *peut* exister des objets gris à l'extérieur du cache. Alors, au moment de récupérer un objet du cache, si ce cache est vide, la valeur de cette variable

est examinée. Si elle est nulle, il n'y a plus d'objet gris, et le marquage termine. Sinon, il faut chercher un objet gris en parcourant le tas.

Le cache est représenté sous la forme d'une pile de pointeurs vers des objets. Cette pile est stockée dans un bloc à part, gray_cache. La variable gray_cache_ptr contient un pointeur vers le sommet de la pile.

Implémentation simplifiée J'ai apporté des simplifications, voire des corrections, à l'implémentation proposée ci-avant. Ces simplifications sont notamment d'ordre structurel et arithmétique.

J'ai *inliné* la procédure find_first_gray_block : j'ai remplacé l'appel de fonction directement par son corps. En effet, chaque appel de fonction entraîne l'allocation d'un bloc de pile suivie de sa libération, ce qui requiert des preuves d'invariance supplémentaires. Or, la fonction find_first_gray_block, qui cherche le premier objet gris dans le tas, ne modifie pas elle-même la mémoire, mais le fait de créer une procédure entraînera l'allocation et la libération d'un bloc de pile, modifications tout à fait inutiles. Comme l'appel à cette procédure n'apparaît qu'une seule fois dans le code, et que le pointeur de pile n'est pas utilisé, on peut donc l'inliner.

La couleur d'un objet est stockée dans son en-tête. Mais on remarquera que la macro Size_header proposée par Xavier Leroy pour lire la taille ne correspond pas à celle décrite en Section 2. En effet, dans l'implémentation proposée, un *décalage* de bits est appliqué. Or, il est difficile de raisonner avec les décalages, notamment il est difficile de décrire ce qu'il se passe lorsqu'un décalage à gauche (ajout de zéros en bits de poids faible) est suivi d'un décalage à droite (ajout de zéros en bits de poids fort) du même nombre de bits, et vice-versa. Par souci de simplifier la preuve, j'ai choisi dans ce stage de ne procéder que par *masquage* de bits et non par décalage. Alors, j'ai construit la structure des en-têtes telle que décrite en Section 2 : les deux bits de poids faible stockent la nature de l'objet (données, pointeurs ou fermeture), les 30 bits de poids faible dont les deux bits de poids faible masqués à zéro indiquent la taille de l'objet, et les deux bits de poids fort sont réservés à une utilisation future. C'est ainsi que, en l'occurrence, ces deux bits de poids fort sont utilisés par l'algorithme de marquage pour stocker la couleur de l'objet.

J'ai également choisi de ne pas représenter gray_cache_ptr comme un pointeur, mais seulement comme un offset, gray_cache_offset ; ainsi, là où on utilisait le pointeur gray_cache_ptr auparavant, on utilisera désormais l'expression "gray_cache"+int32["gray_cache_offset"], bien définie en arithmétique de pointeurs. Ceci permet de simplifier les preuves étant donné qu'on n'a plus besoin de montrer que le pointeur gray_cache_ptr fait toujours référence au bloc du cache. De plus, le test d'égalité entre pointeurs ne s'effectue que si les pointeurs désignent des cases mémoire valides, ce qui peut ne pas être le cas si on utilise de l'arithmétique de pointeurs. Notamment, la sémantique de la condition suivante :

```
(cache == "gray_cache" + GRAY_CACHE_SIZE)
```

n'est pas bien définie, car `"gray_cache" + GRAY_CACHE_SIZE` n'est pas un pointeur valide (il pointe vers la première case *après* le cache, dans le bloc de cache).

J'ai également simplifié les tests de comparaison, notamment dans les boucles. Intuitivement, une boucle for en C est écrite comme suit :

```
for (int i = 0; i < bound; ++i) {
 foo (i);
}
```

Cependant, comme on raisonne sur les entiers 32 bits, il se peut que le calcul de la borne bound produise un dépassement de capacité. Cet *overflow* se traduit alors par un passage dans les négatifs. Dans ce cas, la boucle ne sera pas exécutée, ce qui n'est pas nécessairement ce qu'on veut. En réalité, le test de comparaison doit être remplacé par un test d'égalité.

```
for (int i = 0; i != bound; ++i) {
 foo (i);
}
```

On obtient alors des programmes Cminor légèrement modifiés. C'est à partir de ces programmes Cminor que j'ai raisonné.

```
var "heap_block"[4]
var "heap_start_offset"[4]
var "heap_end_offset"[4]
#define GRAY_CACHE_SIZE 65536
var "gray_cache"[GRAY_CACHE_SIZE]
var "gray_cache_offset"[4]
var "gray_cache_overflow"[4]
#define KIND_RAWDATA 0
#define KIND_PTRDATA 1
#define KIND_CLOSURE 2
#define Kind_header(h) ((h) & 3)
#define COLOR_WHITE 0
#define COLOR_GRAY 0b01000000000000000000000000000000
#define COLOR_BLACK 0b11000000000000000000000000000000
#define Color_header(h) ((h) & 0b11000000000000000000000000000000)
#define Size_header(h) ((h) & 0b00111111111111111111111111111100)
```

Les macros `#define` sont traitées par un préprocesseur, qui effectue les substitutions désirées directement dans le code.

FIG. 43 – Algorithme de marquage simplifié : variables globales et macros

```
"mark_block"(b): int -> void
{
  stack 0;
  var header, cache;
  if (b == 0) return;
  header = int32[b - 4];
  if (Color_header(header) != COLOR_WHITE) return;
  if (Kind_header(header) == KIND_RAWDATA) {
    /* Set it to black now, as there are no pointers within */
    int32[b - 4] = header | COLOR_BLACK;
  } else {
    int32[b - 4] = header | COLOR_GRAY;
    /* Is there room in the gray_cache? */
    cache = int32["gray_cache_offset"];
    if (cache == GRAY_CACHE_SIZE) {
      int32["gray_cache_overflow"] = 1;
    } else {
      int32["gray_cache" + cache] = b;
      int32["gray_cache_offset"] = cache + 4;
    }
  }
}
```

FIG. 44 – Algorithme de marquage simplifié : coloration d'un élément en gris

```
"gc_mark"(root) : int -> void
{
  var numroots, p, cache, b, header, firstfield, n, heap, hs, he;
  heap = int32["heap_block"];
  hs = int32["heap_start_offset"];
  he = int32["heap_end_offset"];
  int32["gray_cache_offset"] = 0;
  int32["gray_cache_overflow"] = 0;
  {{ loop {
    if (root == 0) exit;
    numroots = int32[root + 4];
    p = root + 8;
    {{ loop {
      if (numroots == 0) exit;
      "mark_block"(int32[p]) : int -> void;
      p = p + 4;
      numroots = numroots - 1;
    } }}
    root = int32[root];
  } }}
  {{ loop {
    /* Find next gray object to work on */
    cache = int32["gray_cache_offset"];
    if (cache != 0) {
      cache = cache - 4;
      b = int32["gray_cache" + cache];
      int32["gray_cache_offset"] = cache;
    } else {
      if (int32["gray_cache_overflow"] == 0) exit;
      b = hs;
      {{ loop {
       if (b == he) {exit 1; }
       header = int32[heap + b];
       if (Color_header(header) == COLOR_GRAY) {b = heap + b + 4; exit 0;}
       b = b + 4 + Size_header(header);
      } }}
    }
    /* b is a gray object of kind PTRDATA or CLOSURE */
    header = int32[b - 4];
    int32[b - 4] = header | COLOR_BLACK;
    /* Call mark_block on all (pointer) fields of b.
       Process fields from last to first since this results
       in better gray_cache utilization in case of right-oriented
       data structures such as lists */
    if (Kind_header(header) == KIND_CLOSURE) {firstfield = 4;} else {firstfield = 0;}
    {{ loop {
      if (n == firstfield) exit;
      n = n - 4;
      "mark_block"(int32[b + n]) : int -> void;
    } }}
  } }}
}
```

FIG. 45 – Algorithme de marquage simplifié

3.1.2 Les algorithmes en Coq

En Coq, la sémantique de Cminor est formalisée par des prédicats inductifs. Ceux-ci sont très lourds à manipuler, car leur définition est dirigée par la syntaxe de Cminor. Donc, pour chaque construction de Cminor, il faut inverser le prédicat indiquant sa sémantique.

Or, l'algorithme de marquage ne génère aucune trace d'événements, car aucune procédure externe n'est appelée. De plus, aucune valeur n'est renvoyée en résultat de la procédure de marquage. Ce qui nous intéresse donc est l'évolution de la mémoire durant l'exécution de cette procédure.

Il est alors plus simple de décrire l'évolution de la mémoire directement en Coq, plutôt qu'en Cminor. Ainsi on pourra bénéficier des théorèmes présents dans le modèle mémoire. En effet, dans CompCert, il y a très peu de théorèmes utiles pour raisonner sur la sémantique de Cminor.

De plus, lorsque les algorithmes distinguent plusieurs cas, il est beaucoup plus facile d'analyser ces cas en Coq, car pour chaque cas, les termes sont transformés automatiquement par les règles de réduction de Coq : en quelque sorte, le programme est « exécuté pas à pas » par Coq.

Les algorithmes sont présentés dans le module Marksweep_concrete.

La procédure Cminor mark_block est représentée par un seul algorithme en Coq.

En revanche, la procédure Cminor gc_mark admet des boucles. Pour pouvoir raisonner en Coq sur ces boucles, il faut les considérer isolément. C'est pourquoi cette procédure est découpée en plusieurs parties :

- mark_root_list marque en gris les racines d'une liste de racines
- mark_all_roots marque en gris les racines d'une liste de liste de racines
- mark_children marque en gris les objets fils d'un objet
- find_first_gray_block cherche un objet gris dans le tas
- mark_gray est la boucle de marquage « principale », c'est-à-dire qui s'exécute une fois que les racines ont été marquées en gris (correspondant donc aux étapes 2 à 7 de l'algorithme de marquage décrit page 48).

Par souci d'économie de place, ils ne sont pas présentés ici. On se référera aux sources, prochainement disponibles sur le site [Ram07].

Ces parties sont des termes Coq, et non Cminor : ils ne correspondent pas à des appels de fonction Cminor, donc ils n'impliquent pas d'allocation de bloc de pile.

J'ai montré, dans le module Marksweep_concrete, que, si ces algorithmes terminent, alors on peut construire dans Set la mémoire finale obtenue après exécution de l'algorithme.

3.1.3 Correspondance des algorithmes avec les programmes Cminor

On peut montrer alors que les programmes Cminor et les algorithmes Coq effectuent les mêmes opérations mémoire. Je l'ai fait au moins pour mark_block, dans le module Marksweep_concrete_cminor. Plus exactement, j'ai prouvé que :

1. si à partir d'une mémoire m le programme Cminor termine et renvoie une mémoire m', alors l'algorithme Coq termine et renvoie la même mémoire m'. Donc, si un résultat de correction est vrai au niveau de l'algorithme Coq, il le sera également au niveau de l'algorithme Cminor, puisque les mémoires de sortie sont les mêmes.

2. si à partir d'une mémoire m l'algorithme Coq termine et renvoie une mémoire m', alors le programme Cminor termine et renvoie la même mémoire m'

Tous les théorèmes de correction et de terminaison sont énoncés et prouvés pour les algorithmes Coq seulement. C'est pourquoi il faut montrer les *deux* sens de l'implication. Le sens 1 permet de prouver la *correction* des programmes Cminor à partir de la correction des programmes Coq, tandis que le sens 2 permet de prouver leur *terminaison*.

Le sens 1 se prouve en utilisant la tactique eapply. Il s'agit de construire le terme de preuve correspondant à l'exécution du programme Cminor pas à pas. eapply permet de deviner la valeur de retour et la trace d'exécution du programme Cminor. Par contre, il faut quand même donner le théorème à utiliser, en fait le constructeur du terme de preuve de la sémantique, par exemple lorsqu'on doit prouver qu'une instruction conditionnelle (if) s'exécute, il faut choisir quelle branche du test va s'exécuter. De tels choix sont difficiles à automatiser.

Pour prouver le sens 2, on a besoin d'inverser les prédicats correspondant à la sémantique du programme source Cminor. Lorsqu'on inverse ces prédicats, on obtient tous les cas possibles d'exécution du programme Cminor. On peut automatiser ce processus d'inversion en écrivant une tactique dans le langage de définition de tactiques $\mathcal{L}$tac [BC04] de Coq. La tactique run permet de pratiquer toutes les inversions possibles, éclatant le but à prouver en chacun des cas d'exécution possibles. Elle agit en effet comme si le programme Cminor était exécuté pas à pas, d'où son nom.

```
Ltac invclear H := inversion  H ; clear H ; subst.

Ltac run0 := match goal with
| H : eval_expr _ _ _ _ _ ?e _ _ _ |- _ =>
  match e with
    | Cmconstr.and _ _ => unfold Cmconstr.and in H ; invclear H
    | Cmconstr.or _ _ => unfold Cmconstr.or in H ; invclear H
    | Evar _ => invclear H
    | Eop _ _ => invclear H
    | Eload _ _ _ => invclear H
    | Estore _ _ _ _ => invclear H
    | Ecall _ _ _ => invclear H
    | Econdition _ _ _ => invclear H
    | Elet _ _ => invclear H
    | Eletvar _ => invclear H
    | Ealloc _ => invclear H
  end
| H : exec_stmt _ _ _ _ ?stmt _ _ _ _ |- _ =>
  match stmt with
    | Cmconstr.ifthenelse _ _ _ => unfold Cmconstr.ifthenelse in H ; invclear H
    | Sseq _ _ => invclear H
    | Sreturn _ => invclear H
    | Sskip => invclear H
    | Sexpr _ => invclear H
    | Sassign _ _ => invclear H
    | Sifthenelse _ _ _ => invclear H
    | Sblock _ => invclear H
    | Sexit _ => invclear H
    | Sswitch _ _ _ => invclear H
  end
| H : eval_condexpr _ _ _ _ _ ?c _ _ _ |- _ =>
  match c with
    | CEtrue => invclear H
    | CEfalse => invclear H
    | CEcond _ _ => invclear H
    | CEcondition _ _ _ => invclear H
  end
| H : eval_exprlist _ _ _ _ _ Enil _ _ _ |- _ => invclear H
| H : eval_exprlist _ _ _ _ _ (Econs _ _) _ _ _ |- _ => invclear H
end.
Ltac run := repeat run0.
```

FIG. 46 – Tactique d'inversion d'un prédicat donnant la sémantique d'un programme en Cminor (run). On filtre sur l'arbre de syntaxe abstraite de Cminor.

Dans chaque but généré, on a alors des hypothèses sur les environnements, sous la forme d'égalités. La tactique dereq permet alors de dériver toutes les égalités possibles et de substituer les variables suivant ces égalités. Lors de cette substitution, les hypothèses sont simplifiées, et on peut continuer d'exécuter le programme avec la tactique run.

```
Ltac redenv_hyp :=
match goal with
| H : (PTree.set ?a _ _) ! ?a = _ |- _ => rewrite PTree.gss in H
| H : (PTree.set ?a ?v ?e) ! ?b = _ |- _ => rewrite (PTree.gso (i := b) (j := a) v e) in H ; auto
end.


Ltac injopt := match goal with
| A : Some _ = Some _ |- _ => generalize (option_inj A) ; clear A ; intro A ; endsubst
end.

Ltac dereq0 := match goal with
| A : ?a = ?a |- _ => clear  A
| A : Some _ = Some _ |- _ => injopt
| A : ?a = Some _, B : ?a = Some _ |- _ => rewrite A in B ; injopt
| A : ?a = Some _, B : Some _ = ?a |- _ => rewrite A in B ; injopt
| _ => redenv_hyp
end.

Ltac dereq := repeat dereq0.
```

FIG. 47 – Tactique de dérivation d'égalités (dereq), incluant la simplification d'égalités sur les environnements de variables locales (redenv_hyp)

Le script de preuve se déroule bien. Mais au moment de valider la preuve avec la commande Qed, Coq consomme beaucoup de mémoire (de l'ordre du giga-octet), et met beaucoup de temps (de l'ordre de la dizaine, voire de la vingtaine, de minutes, ce qui est comparable au temps mis pour compiler *tout* le backend CompCert). C'est donc que Coq, bien que très puissant au niveau logique, n'est pas adapté du point de vue pratique, car il manque d'optimisations au niveau de la représentation et du typage des termes de preuve.

Cependant, on aurait peut-être pu se contenter de prouver le sens 1, et montrer le sens 2 à partir du sens 1. En effet, comme l'algorithme de marquage n'effectue aucun appel à une procédure externe, la sémantique de son programme Cminor est déterministe : dès lors qu'il termine, sa mémoire est unique. Alors, pour montrer le sens 2 en supposant que l'algorithme Cminor à partir de la mémoire *mem* calcule la mémoire *mem'*, on lance l'algorithme abstrait sur la même mémoire source *mem*, puis on utilise le sens 1 pour obtenir une mémoire *mem"* obtenue à partir de l'algorithme Cminor à partir de *mem*. Comme la sémantique est déterministe, on a alors $mem" = mem'$. Et donc, on a obtenu le sens 2 car *mem"* par construction a été obtenue par

3.2 Le marquage abstrait

L'étape de marquage ne modifie pas la structure du tas : les objets ont toujours les mêmes fils tout au long de l'exécution de l'algorithme. On peut alors modéliser le marquage à un niveau abstrait. C'est le rôle du module Mark_abstract.

3.2.1 Données nécessaires

L'algorithme de marquage abstrait a besoin d'une représentation abstraite du tas et des racines. Cette représentation est assurée par la donnée de :

– un type A pour les objets

- une fonction, valid, qui pour chaque objet dit s'il est *valide*, c'est-à-dire s'il s'agit bien d'un objet du tas. (Ceci permet de considérer uniquement les objets du tas comme un *sous-ensemble* du type A, et non comme tout le type A lui-même. De plus, grâce à cette fonction, être un objet du tas est *décidable*.)
- une liste roots des objets racine
- une fonction children qui à chaque objet associe la liste de ses objets fils
- une fonction raw qui pour chaque objet dit s'il s'agit d'un objet de données (ceci est distinct du fait de n'avoir aucun objet fils)
- une fonction first_gray qui décide s'il existe encore des objets gris dans le tas, et si oui, en exhibe un

```
Variable A : Set.
Variable valid : A -> bool.
Variable eq_A_dec : forall a b : A, {a = b} + {a <> b}.
Variable first_gray : (A -> color) -> option A.
Hypothesis Hfirst_gray_some : forall col a, first_gray col = Some a -> (valid a = true /\ col a = Gray)
Hypothesis Hfirst_gray_none : forall col, first_gray col = None -> forall a : A, valid a = true -> col
```

FIG. 48 – Données nécessaires pour la représentation du tas et des racines dans l'algorithme abstrait.

Le cache est alors modélisé par une liste Coq d'objets. Pour exprimer que le cache est plein, on n'utilise pas le concept de taille, mais il faut donner un type B : alors, le cache sera accompagné d'un terme de ce type modifié uniquement lorsqu'on empile (push) ou dépile (pop) le cache, par la donnée de deux telles fonctions de type BB. La donnée d'une fonction cache_is_full permet alors de contrôler, sur la donnée de l'état accompagnant le cache, si le cache est plein. Ainsi, le fait que le cache soit plein est indépendant des éléments empilés ou dépilés.

```
Variable B : Set.
Variable cache_is_full : B -> bool.
Variable push : B -> B.
Variable pop : B -> B.
```

FIG. 49 – Données nécessaires pour la modélisation du cache plein ou non dans l'algorithme abstrait.

3.2.2 Algorithmes

Les algorithmes de marquage abstrait prennent en entrée et renvoient en sortie un état abstrait composé de :

- une fonction col qui à chaque objet associe une couleur parmi blanc, noir ou gris
- un booléen overflow qui indique si l'algorithme a tenté d'empiler un objet sur le cache alors qu'il était déjà plein, c'est-à-dire s'il peut y avoir des objets gris hors du cache. Ce booléen ne peut pas être remis à false.
- le cache, sous la forme d'une liste d'objets
- un état abstrait cache_full de type B pour exprimer si le cache est plein ou non

Cet état abstrait n'est cependant pas modélisé par un type *record*, car toutes les fonctions ne modifient pas nécessairement tous les « champs » de cet état.

On a alors les algorithmes de marquage abstrait :

- cache_push empile un élément sur le cache, et met à jour l'état abstrait de cache plein. Si le cache était déjà plein, alors il reste tel quel, mais le booléen overflow est mis à true, indiquant que désormais, il peut y avoir des objets gris hors du cache.
- mark_block est la fonction qui marque en gris un objet. Si l'objet n'est pas valide, alors l'état est renvoyé tel quel. Si l'objet est un objet de données, il est marqué en noir. Sinon, il est marqué en gris, et est mis en cache.
- mark_list marque successivement en gris une liste d'objets. Elle peut être utilisée indifféremment avec la liste des objets fils d'un objet, ou la liste des objets racine. Elle correspond en fait à l'exécution de List.fold_left mark_block
- cache_pop recherche un objet gris en le dépilant un élément du cache. Si le cache est vide, alors il faut distinguer selon la valeur du booléen overflow. Si ce booléen est false, alors l'algorithme n'a jamais tenté d'empiler un objet sur un cache déjà plein, donc le cache a toujours pu contenir tous les objets gris. Il n'y en a donc plus. Sinon, il existe *peut-être* un objet gris dans le tas. Il faut alors le calculer en utilisant la fonction first_gray donnée. Si cette fonction ne renvoie aucun objet, c'est qu'il n'y en a plus.
- mark_gray est une boucle qui dépile les objets du cache, les marque en noir et marque en gris leurs fils, jusqu'à épuiser le cache.
- gc_mark marque en gris les objets racines, puis exécute la boucle mark_gray

Par souci d'économie de place, ces algorithmes ne seront pas présentés ici.

À partir de la coloration initiale où tous les objets sont blancs, on veut connaître la coloration finale obtenue par l'exécution de l'algorithme gc_mark, et alors on veut prouver que, dans cette coloration, les objets marqués en noir sont tous les objets utilisés. Mais on veut aussi prouver que l'algorithme de marquage termine, c'est-à-dire que la boucle mark_gray termine.

3.2.3 Correction

L'objectif est de prouver que, si l'algorithme termine, les objets marqués sont les objets accessibles depuis les objets racine.

La notion d'utilisation considérée ici est une notion faible : les chemins ne sont pas donnés explicitement, car on suppose qu'ils ne changent pas au cours de l'algorithme mémoire.

```
Variable A : Set.
Variable valid : A -> bool.
Variable roots : list A.
Variable children : A -> list A.
Variable raw : A -> bool.

Inductive points_to source target : Prop :=
| points_to_intro : raw source = false -> member target (children source) -> valid target = true ->
  @points_to source target.

Inductive used : A -> Prop :=
| used_root :
   forall n, member n roots -> valid n = true -> used n
| used_child :
   forall k, used k -> forall n, points_to k n ->
   used n
.
```

FIG. 50 – Prédicats abstraits d'accessibilité d'un objet, sans donnée explicite du chemin.

D'abord, on prouve que tous les objets utilisés sont non blancs à la fin de l'algorithme (correction). Pour cela, on a besoin de prouver l'invariant fondamental qu'un objet noir ne pointe jamais vers un objet blanc.

Puis, on prouve que les objets non blancs sont utilisés (complétude). Pour cela, il faut prouver que :

- le cache est une liste sans doublons (ceci afin qu'un même élément ne puisse pas être dépilé du cache plusieurs fois)
- les éléments gris ne sont jamais des objets de données (car ceux-ci sont directement coloriés en noir sans passer par le cache)
- tous les éléments du cache sont gris

Enfin, il est facile de prouver qu'il n'y a plus d'objet gris lorsque l'algorithme de marquage termine. Donc, on a prouvé que, *si* l'algorithme de marquage termine, alors les objets noirs sont exactement les objets utilisés.

3.2.4 Terminaison

A priori, le type A des objets est quelconque. Donc, la terminaison de la boucle mark_gray et donc de l'algorithme de marquage n'est pas assurée. Il faut alors rajouter une hypothèse supplémentaire sur A.

En fait, il suffit de dire que l'ensemble des objets valides est fini. Cependant, pour quelle notion de finitude ? L'idée est de dire que l'ensemble des objets valides est inclus dans une liste.

Alors, le module Mark_abstract_finite montre que sous cette hypothèse, l'algorithme de marquage termine. La preuve de terminaison n'est pas fondée sur la structure de la liste elle-même, mais sur le calcul d'un *poids* qui décroît strictement à chaque tour de la boucle mark_gray. Ce poids est la somme du nombre d'objets *valides* blancs ou gris. Pour pouvoir calculer cette somme, il faut pouvoir parcourir l'ensemble des objets valides. C'est précisément la raison d'être de la liste.

Ce poids décroît strictement, car :

- à chaque tour de boucle mark_gray, il y a toujours au moins un objet gris qui est colorié en noir : celui qui est dépilé du cache
- un objet noir reste toujours noir

Et alors, l'utilisation de la librairie Loop permet de montrer, grâce à cette condition de terminaison, que la coloration finale de l'algorithme de marquage peut être obtenue dans Set.

3.3 Correction et terminaison du marquage concret

Contrairement au marquage abstrait, le marquage concret agit directement sur la mémoire. En plus de montrer que l'algorithme de marquage concret colorie en noir exactement tous les objets utilisés et ceux-là seulement, il faut donc aussi montrer que la structure de tas est conservée, avec les mêmes réalisations de chemins, et les mêmes données pour les objets de données. Ces preuves sont l'objet du module Marksweep_concrete_proof.

3.3.1 État mémoire abstrait

L'idée est de faire correspondre la mémoire concrète avec un état mémoire abstrait modélisant les couleurs et le cache.

L'état mémoire abstrait used_abstraction_param, avec la relation d'abstraction used_abstraction, donnent la structure du tas et des objets racine. Or, le programme Cminor implémentant le GC admet des variables globales. Celles-ci sont donc associées à des blocs, dont il faut assurer qu'ils sont disjoints du tas et des endroits où les racines sont stockées. La donnée des blocs associés à ces variables globales est alors un objet de type record gc_invariants et le prédicat associé marksweep_gc_inv (non détaillés ici), qui inclut la relation d'abstraction used_abstraction.

À cet état mémoire abstrait, il faut rajouter :

- un booléen gray_cache_overflow_bool, vrai si tous les objets gris sont nécessairement dans le cache, faux si le cache a déjà tenté de recevoir un objet alors qu'il était plein, et alors il peut y avoir des objets gris hors du cache
- la valeur gray_cache_offset_value de la variable globale gray_cache_offset, indiquant l'offset où écrire la prochaine valeur à empiler dans le bloc du cache
- une fonction marksweep_col qui à chaque objet du cache associe sa couleur
- le cache sous la forme d'une liste de pointeurs (certains pouvant être nuls)

```
Record mark_abstraction_param : Set := make_mark_abstraction_param {
 gray_cache_overflow_bool : bool;
 gray_cache_offset_value : int;
 marksweep_col : option (block * int) -> Mark_abstract.color;
 marksweep_cache : list (option (block * int))
}.
```

FIG. 51 – Informations sur les couleurs et le cache pour compléter l'état mémoire abstrait

La donnée de ces éléments forme le type record mark_abstraction_param. marksweep_col, gray_cache_overflow_bool, marksweep_cache et gray_cache_offset_value correspondant exactement aux « champs » de l'état mémoire abstrait manipulé par l'algorithme de marquage abstrait, respectivement col, overflow, cache et cache_full. En effet :

- le type A des objets est le type option(block*int) des pointeurs abstraits (None pour le pointeur nul Vint zero, Some (b, i) pour un pointeur Vptr b i)
- le type B de l'indicateur de cache plein est le type int, celui de l'offset du sommet de cache. Le cache est plein si et seulement si cet offset est égal à la taille GRAY_CACHE_SIZE du cache. Empiler (respectivement dépiler) un objet du cache ajoute (respectivement soustrait) 4 à cet offset.

Alors, cet état mémoire abstrait est relié à la mémoire via la relation d'abstraction mark_abstraction, qui requiert également l'état abstrait de la mémoire used_abstraction_param correspondant à la structure du tas et des racines.

```
Record mark_abstraction
  (m : mem) (mp : memory_param) (up : used_abstraction_param) (gp : gc_invariants_param)
  (p : mark_abstraction_param) : Prop :=
 mark_abstraction_intro
 {
  gray_cache_overflow_defined : Some (Vint (if gray_cache_overflow_bool p then one else zero)) =
load Mint32 m gray_cache_overflow_block 0;

 gray_cache_offset_defined :
   Some (Vint (gray_cache_offset_value p)) = load Mint32 m gray_cache_offset_block 0;
 gray_cache_offset_lower_bounded : cmp Cle zero (gray_cache_offset_value p) = true;
 gray_cache_offset_higher_bounded : cmp Cle (gray_cache_offset_value p) GRAY_CACHE_SIZE = true;
 gray_cache_offset_modulo_four : modulo_four zero (gray_cache_offset_value p);

 col_black_a_c : forall b, member b (used_blocklist up) ->
 marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.Black ->
forall v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_BLACK;

 col_black_c_a : forall b, member b (used_blocklist up) ->
 forall v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_BLACK -> marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.Black;

 col_gray_a_c : forall b, member b (used_blocklist up) ->
 marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.Gray -> forall v,
Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_GRAY;

 col_gray_c_a : forall b, member b (used_blocklist up) ->
 forall v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_GRAY -> marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.Gray;

 col_white_a_c : forall b, member b (used_blocklist up) ->
 marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.White ->
forall v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_WHITE;

 col_white_c_a : forall b, member b (used_blocklist up) ->
 forall v, Some (Vint v) = load Mint32 m (memory_heap mp) (signed (sub b four)) ->
Color_header v = COLOR_WHITE -> marksweep_col p (Some (memory_heap mp, b)) = Mark_abstract.White;

 cache_abstraction : pointer_list m gray_cache_block (sub zero four)
(sub (gray_cache_offset_value p) four) (marksweep_cache p);

 cache_elements_in_heap : forall b i, member (Some (b, i)) (marksweep_cache p) -> b = memory_heap mp;
 cache_elements_in_blocklist : forall b i, member (Some (b, i)) (marksweep_cache p) ->
member i (used_blocklist up);
 cache_elements_not_null : member None (marksweep_cache p) -> False;

 cache_elements_not_raw : forall bi, member bi (marksweep_cache p) -> used_raw up bi = false;
 gray_elements_not_raw : forall bi, marksweep_col p bi = Mark_abstract.Gray -> used_raw up bi = false
}.
```

FIG. 52 – Relation d'abstraction pour les couleurs et le cache (mark_abstraction)

Les prédicats de la relation d'abstraction sont les suivants :

– la valeur de la variable globale gray_cache_overflow est 0 (resp. 1) si le booléen gray_cache_overflow_bool

est false (resp. true)
- la valeur de la variable globale gray_cache_offset est donnée par gray_cache_offset_value
- cette valeur est positive
- cette valeur est inférieure ou égale à la taille GRAY_CACHE_SIZE du cache
- cette valeur est multiple de 4
- pour tout objet o du tas, si la couleur abstraite donnée par marksweep_col o est noire (resp. grise, blanche), alors la couleur lue dans l'en-tête de l'objet o est noire (resp. grise, blanche)
- pour tout objet o du tas, si la couleur lue dans l'en-tête de l'objet o est noire (resp. grise, blanche), alors la couleur abstraite donnée par marksweep_col o est noire (resp. grise, blanche)
- le cache est obtenu en lisant la liste de pointeurs dans le bloc du cache, jusqu'à l'offset limite -4 (i.e. il faut lire jusqu'à l'offset 0 inclus). Ici, on utilise pointer_list, le même prédicat que pour lire les objets fils d'un objet du tas
- les éléments du cache sont des pointeurs vers le bloc du tas
- les éléments du cache sont des pointeurs dont l'offset correspond à un objet du tas
- le pointeur nul n'est pas dans le cache
- il n'y a aucun objet de données dans le cache
- les objets gris ne sont pas des objets de données

L'algorithme de marquage abstrait requiert également une fonction qui définit si un objet est valide. Cette fonction est calculée grâce à la seule structure du tas et des données : les pointeurs valides sont les pointeurs vers les objets du tas.

En résumé, l'état mémoire abstrait est donc constitué de :

- un objet de type used_abstraction pour modéliser la structure du tas et des racines[8], relié à la mémoire par la relation d'abstraction used_abstraction
- un objet de type marksweep_gc_inv donnant les blocs des variables globales, et le prédicat gc_invariants assurant que ces blocs sont distincts du tas et des blocs où sont stockés les pointeurs vers les racines
- un objet de type mark_abstraction_param donnant les couleurs, le cache, l'offset de son sommet et le booléen overflow, relié à la mémoire par la relation d'abstraction mark_abstraction

Pour chaque étape de l'algorithme de marquage, il faut alors montrer que la mémoire est reliée à son état mémoire abstrait par les différents prédicats d'abstraction.

L'état mémoire used_abstraction_param du tas et des racines ne change pas, sauf éventuellement le booléen used_extra_zero qui indique si les deux bits de poids fort réservés dans les en-têtes des objets du tas sont nuls. Initialement à true avant l'algorithme du marquage, ce booléen devient false durant cet algorithme afin que la gestion de ces deux bits de poids fort soit rendue possible par le prédicat mark_abstraction. En effet, ces deux bits de poids fort sont utilisés pour stocker la couleur des objets.

Les blocs des variables globales donnés par l'objet de type marksweep_gc_inv ne changent pas non plus, car la sémantique de Cminor interdit leur modification.

L'état mémoire abstrait mark_abstraction_param, quant à lui, change. Il est donné par l'algorithme de marquage abstrait, exécuté parallèlement au marquage concret. Il faut alors montrer que chaque étape conserve la relation d'abstraction conformément aux exécutions en parallèle des deux algorithmes.

[8] ainsi qu'un objet de type memory_param donnant le bloc du tas et les bornes de la zone du tas, ainsi qu'un pointeur vers la première liste de racines

3.3.2 mark_block

mark_block est la brique de base de l'algorithme de marquage. Toutes les autres étapes de l'algorithme de marquage utilisent mark_block. Il faut donc prouver une fois pour toutes la conservation des relations d'abstraction.

Comme c'est une procédure Cminor, il faut allouer une pile d'exécution. Cette allocation réussit toujours. On montre alors que le bloc de pile alloué est distinct du tas, des blocs où sont stockés les racines, ainsi que des blocs des variables globales. En effet, avant l'allocation, ces blocs sont valides, contrairement au bloc de pile.

Et alors, comme le modèle mémoire fournit le théorème énonçant que les blocs distincts du bloc de pile sont inchangés, on peut alors prouver que les relations sont conservées par l'allocation avec les mêmes états abstraits.

Cependant, à la fin de l'exécution du corps de la procédure, le bloc de pile est libéré. Il faut donc prouver que les relations sont conservées par la libération avec un état abstrait *a priori* différent de l'état abstrait initial avant l'allocation, mais inchangé par la libération. Ceci s'effectue également en remarquant, grâce à un théorème fourni par le modèle mémoire, que les blocs différents du bloc libéré sont inchangés.

On peut alors se focaliser sur le corps de la procédure.

On suppose que les relations d'abstraction sont vérifiées sur la mémoire initiale m et que l'algorithme reçoit en entrée un pointeur nul ou valide (désignant un objet du tas).

Remarquons que l'algorithme effectue une lecture mémoire après avoir écrit : il colorie un objet en gris, puis il lit en mémoire la valeur donnée par la variable globale indiquant l'offset du sommet du cache. Il faut montrer qu'on peut permuter ces lectures et écritures et effectuer toutes les lectures dans la mémoire initiale. On a besoin pour cela de deux théorèmes :

- l'écriture dans un bloc laisse invariante la lecture dans un bloc différent. Ce théorème est donné par le modèle mémoire, il s'agit en fait du théorème fondamental justifié par la structure de blocs de la mémoire.
- l'écriture dans un bloc d'un chunk Mint32 (4 octets) laisse invariante la lecture d'un chunk Mint32 dans le même bloc, mais à un offset distinct et *congru modulo 4* à l'offset de l'écriture. Ce théorème n'est pas prévu par le modèle mémoire, il faut donc le prouver à la main (ce qui est réalisé dans le module Memory), à l'aide de la librairie Int_addons que j'ai développée, fournissant un théorème énonçant que deux entiers 32 bits a et b distincts mais congrus modulo 4 sont tels que les intervalles $a..a+4$ et $b..b+4$ sont disjoints. Ce théorème permet alors (moyennant la conversion des entiers 32 bits vers leur représentation signée) d'utiliser un théorème fourni par le modèle mémoire, énonçant que l'écriture dans un bloc à l'offset a d'un chunk de n octets laisse invariante la lecture d'un chunk de p octets dans le même bloc à l'offset b, si a..a+n et b..b+p sont disjoints.

Alors, on peut prouver que l'algorithme concret de marquage fait les mêmes écritures mémoire que l'algorithme modifié où les lectures se font dans la mémoire initiale. Ce dernier algorithme ne correspond pas *a priori* à une instruction Cminor (c'est-à-dire qu'il n'existe pas d'instruction Cminor telle que la sémantique se traduise à l'opération près vers l'algorithme modifié).

Ainsi, tous les lemmes d'invariance sont prouvés sur l'algorithme modifié.

On montre d'abord que l'algorithme se déroule correctement (toutes les lectures et écritures se passent bien) et renvoie une mémoire qu'on peut calculer dans Set.

On peut alors supposer l'existence d'une mémoire *m'* renvoyée par l'algorithme, et prouver les propriétés d'invariance entre m et *m'*.

- Un bloc valide dans m est valide dans m'
- Un offset valide dans un bloc de m est aussi valide dans le même bloc de m'
- Les blocs différents du cache, du tas, et des variables globales stockant l'offset de sommet du tas et la variable overflow, ont leur contenu inchangé entre m et m'
- Si le pointeur fourni en paramètre de l'algorithme n'est pas nul, il désigne un objet du tas (par hypothèse), et alors son en-tête est modifiée par un changement de couleur : noir si c'est un objet de données, gris sinon. Mais la taille et la nature de l'objet restent les mêmes.
- Si le pointeur fourni en paramètre de l'algorithme n'est pas nul, il désigne un objet du tas (par hypothèse), et alors les en-têtes des autres objets sont inchangées.
- Et alors, pour chaque objet o du tas de m, l'en-tête est lisible à l'offset o-4 du tas de m'
- Les tailles et les natures de tous les objets du tas sont inchangés.
- Ce dernier lemme permet de prouver que la liste des objets du tas est inchangée.
- Les données de tous les objets du tas sont inchangés.
- Ce dernier lemme permet de prouver que l'abstraction du tas et des racines de la mémoire est conservée.

Enfin, on montre que l'abstraction de la couleur et du cache est conservée avec le nouvel état abstrait donné par l'exécution de l'algorithme abstrait mark_block. Cette preuve est extrêmement longue (plusieurs centaines de tactiques), notamment pour prouver l'abstraction du cache. Cette abstraction se prouve en utilisant un théorème dérivé de l'invariance de la liste des pointeurs (voir pointer_list).

Pour tous ces théorèmes, il faut prouver que les zones d'invariance sont disjointes des zones écrites. Ces propriétés de disjonction rendent les preuves ardues du point de vue arithmétique. Et alors, on utilise les théorèmes d'invariance du tas, de la liste de pointeurs, de lia liste de racines...

Et alors, grâce aux lemmes de conservation prouvés pour l'allocation d'un bloc de pile et sa libération, la procédure mark_block hérite de la conservation des abstractions pour le corps. Elle laisse inchangés les blocs mémoire inchangés par son corps s'ils sont distincts du bloc de pile.

3.3.3 Déroulement de la preuve du marquage

Muni de la preuve de correction de mark_block, on peut alors prouver une à une les différentes étapes du marquage.

find_first_gray_block recherche un objet gris dans le tas, si le cache est vide mais qu'il peut y avoir des objets gris hors du cache (overflow = true). Ce fragment de l'algorithme ne modifie pas la mémoire. On en écrit une version abstraite, first_gray, à utiliser avec l'algorithme abstrait de marquage.

Puis, on montre que les réponses (objet gris envoyé, ou preuve qu'il n'y a pas d'objet gris dans le tas) de ces deux algorithmes se correspondent. Cette preuve s'effectue par récurrence sur une liste qu'on suppose être un suffixe de la liste des objets du tas. Cette liste correspond à la partie du tas comprise entre l'offset à partir duquel find_first_gray_block doit chercher l'objet gris (c'est-à-dire que depuis le début du tas et jusqu'à cet offset, il n'y avait aucun objet gris), et la fin du tas.

Il faut ensuite montrer que, si on dépile un objet du cache, alors les relations d'abstraction sont conservées avec le nouveau cache. Là encore, cette preuve est longue à cause de la preuve d'invariance de la queue du cache. Grâce aux invariants, présents dans la relation d'abstraction, que tous les objets du cache sont gris, on montre alors que l'élément dépilé du cache est gris.

On prouve ensuite que le fait de colorier un objet gris en noir conserve la relation d'abstraction. Cette preuve utilise les théorèmes d'invariance, les en-têtes des blocs n'étant pas modifiées sauf pour l'objet colorié, dont la nature et la taille ne changent pas.

On prouve ensuite simultanément la terminaison et la correction de mark_children, qui marque en gris les objets fils d'un objet gris. On prouve que mark_children termine avec une nouvelle mémoire ayant les propriétés d'abstraction avec le nouvel état abstrait, puis, comme mark_children est fonctionnelle, on en déduit la correction. Cette preuve utilise la preuve de correction de mark_block. Elle se déroule par récurrence sur une liste qu'on suppose être un suffixe de la liste des objets fils de l'objet considéré. Elle utilise un théorème énonçant que si deux mémoires sont telles que les listes de pointeurs lues entre deux offsets sont les mêmes, alors la lecture d'un suffixe de cette liste de pointeurs renvoie la même liste entre les deux mémoires.

Ces quatre preuves permettent de montrer la correction du corps de la boucle mark_gray_body, en recollant les morceaux. On peut alors prouver que la boucle mark_gray est correcte sous réserve qu'elle termine. Mais on prouve aussi que si l'algorithme abstrait mark_gray termine, alors l'algorithme concret mark_gray termine.

On prouve ensuite la terminaison puis la correction de mark_root_list. Ces preuves s'effectuent par récurrence sur une liste lue avec root_list sur des cases mémoire qui sont des positions où des pointeurs vers des racines sont stockés. Or, ces positions et leur contenu sont laissés invariants par mark_block. Donc, la récurrence passe, la queue de la liste est lue avec root_list aussi sur la mémoire obtenue après avoir marqué en gris la tête de la liste.

La terminaison et la correction de mark_all_roots s'effectue de manière analogue.

On peut alors montrer la correction de l'algortihme de marquage tout entier gc_mark, en recollant les morceaux. Mais il faut que tous les objets du tas soient blancs au départ. Ceci est vrai si on suppose au départ que les deux bits de poids fort réservés dans les en-têtes des objets du tas sont égaux à zéro.

La terminaison de gc_mark se déduit de celle de mark_gray. Ele découle donc de la terminaison de l'algorithme abstrait correspondant, qu'on montre en fournissant la liste des objets du tas comme liste contenant tous les objets valides.

Et alors, la relation d'abstraction permet de montrer que, à la fin de l'algorithme de marquage, les objets noirs sont exactement les objets utilisés, et les objets blancs sont exactement les objets libres.

4 Le nettoyage et l'allocation

Il peut sembler incongru de présenter dans une même section nettoyage et allocation, étant donné que le nettoyage est une étape du mark and sweep alors que l'allocation est largement indépendante du GC.

Cependant, du point de vue de la preuve, des similitudes existent. En effet, l'étape de sweep fait intervenir la coalescence (fusion) de deux blocs libres adjacents, qui est une opération duale de celle de la séparation d'un bloc libre en deux lors de l'allocation. Des problèmes concernant la structure de tas surgissent alors.

Ces preuves étant inachevées, cette section est moins détaillée que les précédentes.

4.1 Le nettoyage

Xavier Leroy a proposé un algorithme de sweep avec coalescence. Voici comment il procède.

- Si on est à la fin du tas, alors le nettoyage termine.
- Sinon, on considère un objet o. Si cet objet est blanc, il est libre. Si l'objet précédemment visité est libre, alors on coalesce o avec lui. Sinon, on transforme o en un objet de données qu'on ajoute à la freelist.
- Si o n'est pas blanc, alors on sait qu'il n'est pas libre. On le remet blanc.
- On avance après la fin de l'objet o

```
"gc_sweep"() : void
{
  stack 0;
  var scan_ptr, scan_end, last_free_block, end_last_free_block,
    header, size;
  last_free_block = "freelist_head";
  end_last_free_block = 0;
  scan_ptr = int32["heap_start"];
  scan_end = int32["heap_end"];
  {{ loop {
    if (scan_ptr >= scan_end) exit;
    header = int32[scan_ptr];
    size = Size_header(header);
    if (Color_header(header) == COLOR_WHITE) {
      /* reclaim this block */
      if (scan_ptr == end_last_free_block) {
        /* coalesce it with last free block */
        int32[last_free_block - 4] =
          int32[last_free_block - 4] + ((size + 4) << 2);
        end_last_free_block = end_last_free_block + size + 4;
      } else {
        /* insert new free block in free list */
        int32[scan_ptr] = header & ~0xF;  /* clear mark and kind bits */
        int32[last_free_block] = scan_ptr + 4;
        last_free_block = scan_ptr + 4;
        end_last_free_block = last_free_block + size;
      }
    } else {
      /* clear mark on this block */
      int32[scan_ptr] = header & ~COLOR_BLACK;
    }
    scan_ptr = scan_ptr + 4 + size;
  } }}
  int32[last_free_block] = 0; /* terminate free list */
}
```

FIG. 53 – Algorithme de nettoyage d'origine en Cminor, erroné

Cet algorithme, cependant, néglige le cas où l'objet o visité est libre mais vide. Dans ce cas, il est quand même ajouté à la freelist, mais il ne pourra pas contenir de pointeur vers l'objet suivant de la freelist. Ce pointeur sera stocké dans les cases mémoire de l'en-tête de l'objet suivant o, ce qui cassera la structure de tas. Par conséquent, ce programme est bogué.

Il faut donc prendre garde à ne pas ajouter d'objets vides dans la freelist. Mais pour autant, l'objet vide ne doit pas être abandonné tout de suite : en effet, il se peut que l'objet suivant soit libre, et dans ce cas, il faudra le coalescer avec l'objet vide.

J'ai écrit directement en Coq un algorithme de nettoyage au niveau de la mémoire. Par souci d'économie de place, je ne l'inclurai pas dans ce rapport ; on se référera aux sources [Ram07] prochainement disponibles.

Cet algorithme procède de la façon suivante.

- Si on est à la fin du tas, alors le nettoyage termine.
- Sinon, on considère un objet o. Si cet objet n'est pas blanc, alors on sait qu'il n'est pas libre. On le remet blanc. Sinon, il est libre.
- Si l'objet précédemment visité est libre, alors on l'ajoute à la freelist s'il était vide auparavant, puis on coalesce o avec lui. Sinon, on ajoute o à la freelist si o n'est pas vide.
- On avance après la fin de l'objet o

Il faut prouver que cet algorithme, s'il est lancé à partir d'une mémoire dont le tas est bien formé et telle que les objets du tas coloriés en blanc soient exactement les objets libres, conserve la structure du tas, ne modifie pas les objets accessibles et intègre les objets libres dans la freelist.

On ne peut pas raisonner par induction sur le prédicat de bonne formation du tas, car celui-ci change. On raisonne donc par récurrence sur une liste l qui représente la partie du tas à visiter entre la position courante et la fin du tas. Il faut alors relier cette liste au tas l avant et après le nettoyage. On sait que cette liste l est suffixe du tas avant itération, et que sa queue est suffixe du tas après itération. Mais il faut pouvoir prouver que le tas après itération est bien formé. Il faudrait alors couper en deux la partie du tas déjà visitée au niveau du dernier objet, qui va subir une fusion avec l'objet suivant ; puis il faudrait alors recoller ensemble ces tas et la queue du tas qu'il reste à visiter.

À l'heure actuelle, l'invariant n'est pas encore finalisé pour faciliter ces preuves de bonne formation du tas : je tente de mener la preuve en même temps que cet invariant est mis au point, mais les tentatives n'aboutissent pas.

Il faudrait probablement développer un module à part, qui montre que la fusion de deux objets de données adjacents produit un tas bien formé.

Or, en plus de ces problèmes structurels se pose un problème arithmétique : les deux valeurs des tailles sont codées sur 30 bits seulement, car les deux bits de poids fort des en-têtes sont réservés. Utiliser un décalage pour stocker les tailles des objets dans les en-têtes ne changerait rien, elles seraient toujours codées sur 30 bits seulement. Comment assurer alors que leur somme sera également codée sur 30 bits ? Ceci est faux en général. En effet, on peut considérer un tas constitué d'un objet de taille maximale $(2^{30} - 4)$, et d'un objet de taille nulle. Fusionner deux tels objets donnerait un objet de taille 2^{30} (car on intègre l'en-tête du second objet aux données du premier objet) nombre codé sur 31 bits impossible à représenter dans l'en-tête d'un objet du tas.

Il faut donc introduire une hypothèse sur le tas. Cette hypothèse doit assurer que la somme de deux objets consécutifs quelconques, plus 4, est positive et inférieure à 2^{30}-1. Mais cette hypothèse doit être invariante par fusion de deux blocs adjacents. Cette invariance impose donc que le tas tout entier soit de taille inférieure à 2^{30}-1. Il faudrait montrer que cette condition est suffisante. Alors, on pourrait montrer que la somme d'un nombre quelconque de tailles d'objets adjacents augmentées de 4 est positive et inférieure à 2^{30}-1, ce qui est suffisant pour la stocker sur 30 bits.

4.2 L'allocation

L'allocation est largement indépendante du GC. Il s'agit d'une allocation en *first-fit* à partir d'une freelist : une liste d'objets libres est parcourue, le premier objet de taille suffisamment grande est pris et coupé en deux, une partie constituant l'objet alloué proprement dit, de taille requise, l'autre partie devant être remise à la freelist.

La preuve de cet algorithme a été amorcée dans le module Memalloc. L'implémentation proposée en Cminor fonctionne de la manière suivante :

- parcourir la freelist jusqu'à trouver un objet dont la taille est plus grande que la taille requise
- si l'objet a exactement la taille requise, c'est lui qui est alloué, il faut alors supprimer cet objet de la freelist en remplaçant le pointeur de l'objet précédent par le pointeur vers l'objet suivant l'objet alloué
- si l'objet a la taille requise + 4, alors on le coupe en deux, créant alors l'objet à allouer, à gauche, et un objet vide à droite ; il faut alors supprimer l'objet de la freelist, car l'objet vide ne doit pas y être intégré
- sinon, l'objet est coupé en deux, l'objet à allouer constituant la partie droite, la partie gauche restant dans la freelist qui demeure inchangée

```
"freelist_alloc"(req_size) : int -> int
{
  stack 0;
  var p, b, header, size, newsize;

  p = "freelist_head";
  {{ {{ loop {
      b = int32[p];               /* b is current free block */
      if (b == 0) exit 2;         /* free list exhausted */
      header = int32[b - 4];
      size = Size_header(header);
      if (size >= req_size) exit;
      p = b;                      /* move to next block */
    } }}
    /* Found a free block large enough */
    if (size == req_size) {
      /* there is nothing left of the free block, remove it
         from free list */
      int32[p] = int32[b];
      return b;
    }
    else if (size == req_size + 4) {
      /* one word remains free, which is too small to put
         on free list.  Do as above, but mark remaining word
         so that it can be coalesced later. */
      int32[p] = int32[b];
      int32[b + req_size] = 0;    /* header with size == 0 color = white */
      return b;
    } else {
      /* cut free block in two:
         - first part remains free --> just reduce its size
         - second part is returned as the free block */
      newsize = size - (req_size + 4);
      int32[b - 4] = newsize << 2;
      return b + newsize + 4;
    }
  }}
  return 0;                     /* free list exhausted */
}
```

FIG. 54 – Allocation (programme Cminor)

L'allocation repose donc sur la structure de freelist. En Coq, la freelist se modélise par une

f

FIG. 55 – Allocation (programme Cminor)

chaîne de pointeurs, la freelist étant constituée de la liste des pointeurs non nuls successivement rencontrés jusqu'à tomber sur le pointeur nul. L'endroit de départ où est stocké le premier pointeur n'est pas comptabilisé. C'est donc une liste chaînée, similaire à l'enchaînement des listes de racines l'une à la suite de l'autre (voir all_roots).

- On lit un pointeur à l'endroit considéré.
- Si le pointeur est nul, alors la liste de pointeurs est vide.
- Sinon, c'est un pointeur Vptr b i. Alors, ce pointeur est ajouté à la liste, et on continue de lire la chaîne à partir de l'offset i du bloc b.

```
Inductive chain (m : mem) : block → int → list (block × int) → Prop :=
| chain_end : ∀ b i, Some (Vint zero) = load Mint32 m b (signed i) →
chain m b i nil
| chain_cont : ∀ b i b' i', Some (Vptr b' i') = load Mint32 m b (signed i) →
∀ l, chain m b' i' l → chain m b i ((b', i') : : l)
.
```

FIG. 56 – Représentation de la freelist sous la forme d'une chaîne de pointeurs (chain)

Dans le cas de la freelist, cette chaîne de pointeurs commence à un bloc spécial, freelist_block. Ce bloc n'est pas un objet de la freelist, mais contient un pointeur vers le premier objet de la freelist. On veut que tous les objets de la freelist soient dans le tas. Un objet de la freelist doit donc posséder le pointeur vers son suivant dans la freelist en tant que première case de ses données. La freelist ne peut donc pas comporter d'objets de taille nulle.

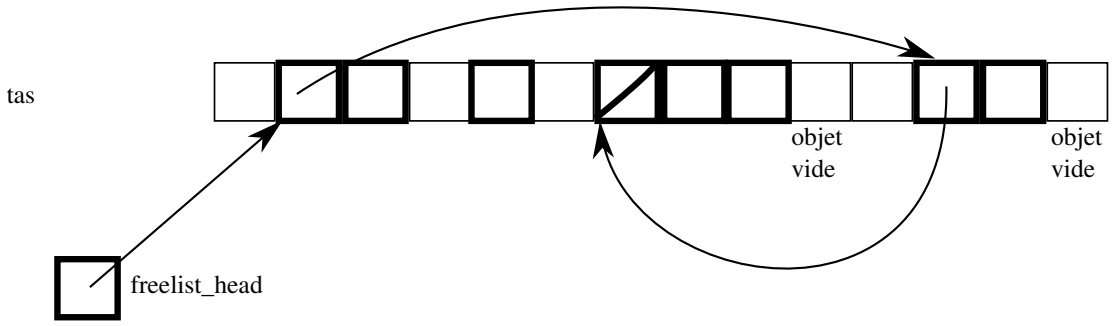

FIG. 57 – Représentation de la freelist comme chaîne d'objets du tas. Les objets vides ne peuvent pas être dans la freelist.

L'algorithme Coq se divise en deux parties : la recherche du bloc libre d'une part, et la division de ce bloc en deux pour l'allocation d'autre part.

La recherche de l'objet libre ne modifie pas la mémoire. Elle se déroule comme suit :

- Lire le pointeur vers l'objet suivant de la freelist. Si ce pointeur est nul, alors la recherche échoue.
- Sinon, il s'agit d'un objet. Lire sa taille dans son en-tête. Si cette taille est plus grande que la taille requise, alors l'objet est trouvé et on sort. Sinon, on continue la recherche.

Elle requiert les hypothèses suivantes sur la mémoire :

- la freelist est bien formée

– chaque objet de la freelist admet une en-tête

Et alors, on montre que si la recherche termine, alors, par induction sur l'exécution de la boucle, soit elle échoue car il n'y a pas d'objet de taille suffisamment grande, soit un objet est trouvé et alors il s'agit d'un objet de la freelist dont la taille est plus grande que la taille requise. On montre la terminaison de la boucle de recherche par induction sur la freelist.

À partir de l'objet libre ainsi trouvé, on veut pouvoir créer l'objet à allouer. Trois cas se présentent :

– l'objet a exactement la taille requise,
– l'objet a la taille requise + 4, l'allocation laissera un objet vide
– l'objet est sensiblement plus grand

Pour que cet algorithme puisse se dérouler correctement, il faut que la mémoire satisfasse les hypothèses suivantes :

– la freelist est bien formée
– le tas est bien formé
– les objets de la freelist sont dans le tas
– les objets de la freelist sont des objets de données dont les deux bits de poids fort sont nuls : l'en-tête contient exactement leur taille
– les objets de la freelist sont de taille non nulle

Alors, on peut montrer que la découpe de l'objet s'effectue correctement. En effet, les opérations d'écriture se passent bien puisque ne sont écrits que l'en-tête de l'objet trouvé et une de ses case de données. Cette preuve de plusieurs centaines de lignes fait intervenir beaucoup d'arithmétique sur les entiers 32 bits et préfigure la forme des preuves sur cet algorithme.

On montre alors que les objets distincts de l'objet libéré sont inchangés (en-tête et données). On montre aussi que les blocs mémoire distincts du tas et du bloc où est stocké le pointeur vers le premier objet de la freelist (bloc de tête de freelist), sont inchangés. Ces derniers théorèmes permettent de montrer, sous les hypothèses que :

– la structure de pointeurs vers les racines est bien formée,
– les positions où sont stockés les pointeurs vers les racines sont dans des blocs distincts du tas et du bloc de tête de freelist
– les objets accessibles par réalisation concrète sur la mémoire de départ sont dans le tas,

que les réalisations concrètes de chemins sont inchangées par l'allocation.

Mais il faut alors montrer que la mémoire d'arrivée vérifie les mêmes hypothèses que la mémoire de départ, sur la structure de la freelist et du tas. Les problèmes arithmétiques surviennent lorsqu'il faut couper un bloc en deux. Car de la même façon, pour montrer que le tas est bien formé, il faut le couper en trois au niveau de l'objet trouvé, sous la forme (A ++ o : : B), puis recoller la partie A avec les deux nouveaux objets o, o' et la partie C. Or, pour intégrer le nouvel objet o', il faut montrer que la fin du tas est distincte de l'offset o'-4 où l'objet trouvé est lui-même coupé en deux.

La séparation et la fusion du tas associées à la gestion des lectures et écritures en mémoire engendrent donc des problèmes arithmétiques rendant la preuve extrêmement fastidieuse, si bien qu'à l'heure actuelle, elle demeure inachevée.

5 Conclusion

5.1 Bilan

J'ai finalement réussi à prouver l'algorithme de marquage, du moins au niveau des algorithmes en Coq au niveau des opérations mémoire, c'est-à-dire au même niveau sémantique que les programmes Cminor. La preuve totale comprenant la formalisation du tas couvre de l'ordre de 10000 lignes de preuve Coq, ce qui représente une taille d'environ 30% inférieure à la preuve de tout le backend Cminor/PowerPC.

Module	Spécifications (lignes)	Preuves (lignes)
Constrpath	21	54
Int_addons	675	885
List_addons	211	492
Loop	82	127
Mark_abstract_finite	155	178
Mark_abstract	557	709
Marksweep_concrete_cminor	112	1000
Marksweep_concrete_proof	1077	3352
Marksweep_concrete	675	598
Memory	2184	2897
TOTAL	**5749**	**10292**
Backend CompCert (par comparaison)	*17558*	*15044*

FIG. 58 – Taille des spécifications et des preuves en nombre de lignes (marquage et librairies supplémentaires)

Cependant, je n'ai pas pu poursuivre plus avant dans la preuve du GC, ayant rencontré de nombreuses difficultés d'ordre technique durant la preuve.

Tout d'abord, la moindre modification du modèle du tas ou des racines implique de revérifier à la main toutes les preuves. Or, ces passages peuvent se révéler fastidieux, notamment à cause des scripts de preuve utilisant des noms d'hypothèses générés par Coq. Modifier les spécifications en amont change l'agencement des hypothèses, à qui Coq attribue alors des noms différents. Or, il est pourtant difficile, notamment pour des tactiques telles que inversion, de prévoir l'agencement des hypothèses afin de leur donner explicitement un nom. De plus, le temps de validation des preuves est très long (il faut près de 5 minutes pour compiler le module Memory sur un Pentium 4 1.5 GHz). Alors, une modification dans le modèle du tas peut prendre une dizaine de minutes pour la revérification des preuves.

J'ai également dû adapter les spécifications elles-mêmes aux contraintes techniques de Coq. En effet, il est arrivé plusieurs fois (block_description_from_to, find_root_list) que Coq boucle au moment de contrôler le type d'un objet défini selon un certain paradigme (par exemple, récurrence Fixpoint sur l'entier naturel représentant le numéro de la liste de racine pour la recherche d'une liste de racines), et qu'il faille choisir un autre paradigme (boucle générique avec la librairie Loop) pour s'en sortir. Ces bogues de Coq sont pourtant difficiles à reproduire sur des exemples élémentaires, ce qui empêche de produire un report de bogue à l'intention des développeurs de Coq.

Mais les difficultés les plus notables qui compliquent les preuves « au quotidien » sont celles apparaissant au niveau arithmétique. En effet, raisonner sur les entiers machine 32 bits n'est pas

aisé, car beaucoup d'assertions intuitives comme la compatibilité de l'ordre sur les représentations signées avec les opérations telles que l'addition, sont fausses. Alors, il semble très difficile de développer des tactiques d'automatisation efficaces pour le raisonnement sur les entiers 32 bits. Un pendant à omega pour

En somme, Coq fournit un niveau de confiance très élevé, qui justifie son choix pour le développement du compilateur certifié CompCert ; cependant, du point de vue pratique, les techniques de preuve sont lourdes et manquent d'automatisation, prix d'une logique (le Calcul des Constructions Inductives) très expressive.

5.2 Travaux reliés

Les preuves de GC sont assez difficiles à formaliser sur machine, car les GC modifient la structure de la mémoire, ces modifications étant très souvent passées sous silence dans les preuves de GC sur papier. Les assistants de preuve sont intransigeants de ce point de vue, ce qui entraîne des preuves extrêmement techniques parfois avec des outils guère adaptés.

Un GC incrémental a été prouvé en Coq et est disponible dans les contributions Coq [CGN03]. Mais il s'agit là de la preuve d'un algorithme abstrait de GC, et non d'une implémentation. Donc, la structure de la mémoire est abstraite : la mémoire est réduite au tas, les pointeurs vers les objets racine étant stockés dans d'autres structures, et les algorithmes sont écrits en Coq au niveau abstrait, et non au niveau du langage d'implémentation.

Cependant, des implémentations de GC ont déjà été vérifiées en Coq, notamment [MSLL07]. Mais celui-ci est écrit en langage assembleur, donc de très bas niveau, ce qui ne convient pas à notre cadre, où le GC doit être écrit dans un langage de niveau moyen destiné à être compilé afin de bénéficier des optimisations proposées par le backend CompCert. La gestion des racines est aussi simplifiée par le fait qu'elles soient stockées directement dans les registres et non en mémoire. De plus, du point de vue arithmétique, les pointeurs sont des entiers naturels quelconques et non des entiers machines, ce qui rend le déroulement des preuves beaucoup plus aisé étant donné que le raisonnement sur les entiers est plus ou moins automatisé. Toutefois, ces travaux ont une portée très importante du point de vue de la compilation certifiée étant donné qu'ils proposent une méthode pour interfacer le GC, l'allocateur et le mutateur afin d'aboutir à la certification d'un gestionnaire de mémoire complet.

5.3 Travaux futurs

Tout d'abord, il faut achever la preuve des étapes de sweep et de l'allocation du GC mark and sweep. Ces preuves donneront une idée de l'utilisabilité de la notion de tas, et requièrent d'étendre la librairie des entiers 32 bits, voire d'y automatiser le raisonnement.

Les structures de la mémoire présentées dans ce stage ne sont pas propres au mark & sweep. En effet, la notion de chemin a été dissociée de son interprétation justement pour pouvoir raisonner sur des structures plus spécifiques comme pour les GC stop & copy, où le chemin doit être interprété à cheval sur deux tas, le tas à copier et le nouveau tas en construction. Tenter de prouver d'autres GC comme le stop & copy, ou même des GC incrémentaux, peut permettre de mettre à l'épreuve les structures développées dans ce stage et de tester leur polyvalence, ou de les généraliser. Mais si on veut pousser encore plus loin en tentant de prouver des GC parallèles, il faut modifier les structures en profondeur et trouver une sémantique parallèle pour Cminor. Les travaux de [ABN$^+$] donnent un début de réponse à ce problème.

Pour que les preuves d'implémentations de GC trouvent leur place au sein du développement du frontend MiniML du compilateur CompCert, il faut également les interfacer avec la preuve de l'allocateur et du mutateur, afin d'obtenir un gestionnaire de mémoire complet qu'on puisse relier à la sémantique du langage MiniML. Il faut pour cela développer un framework afin d'abstraire les GC et leurs preuves, voire d'abstraire complètement le modèle du tas.

5.4 Remerciements

Je remercie l'ensemble de l'équipe Gallium, et en particulier Xavier Leroy, Jean-Baptiste Tristan, Zaynah Dargaye ainsi que Benoît Razet, et Andrew Tolmach, professeur invité, pour leur accueil chaleureux et pour leurs précieux conseils.

Je remercie également ma famille, et en premier lieu mes parents et mes oncles et tantes, qui m'ont beaucoup soutenu pendant ce stage.

A Librairies utilitaires

A.1 Entiers 32 bits : Int_addons

La librairie Integers du compilateur certifié CompCert définit un entier machine 32 bits comme un entier relatif accompagné d'une preuve qu'il est compris entre 0 et 2^{32}. Cet entier relatif constitue la représentation non signée de l'entier machine, obtenue avec la fonction unsigned. La représentation signée, obtenue avec la fonction signed, correspond à soustraire 2^{32} à la représentation non signée si elle est plus grande que $2^{31}-1$. Ainsi, un entier dont la représentation signée est positive a les mêmes représentations signée et non signée. Les opérations arithmétiques sont définies modulo 2^{32} sur les représentations non signées : addition add, multiplication mult... Ce sont les mêmes quelle que soit la représentation, signée ou non signée. En revanche, les opérateurs de comparaison ne sont pas les mêmes en représentation signée qu'en représentation non signée.

Quelques théorèmes de base sont fournis, découlant des propriétés algébriques de Z conservées par le passage modulo 2^{32} : associativité, commutativité, distributivité des opérations.

Comme la sémantique de Cminor interprète les offsets en représentation signée, les opérateurs de comparaison de pointeurs se fondent sur les opérateurs de comparaison en représentation signée. La fonction cmp synthétise ces opérateurs. Elle prend un argument indiquant la comparaison : Cle pour , Clt pour $<$, Cge pour , Cgt pour $>$, et deux entiers 32 bits, et renvoie un booléen selon que la relation de comparaison soit vérifiée ou non.

Cependant, très peu de théorèmes sont fournis concernant ces opérateurs et vis-à-vis des opérations. C'est pourquoi j'ai dû développer une librairie supplémentaire, Int_addons.

Égalité

- Correspondance entre l'opérateur d'égalité eq et le prédicat d'égalité
- Deux entiers ayant les mêmes représentations signées sont égaux (la librairie Integers ne prévoit le résultat que pour les représentations non signées, résultat découlant de la définition)

Opérateurs

- $a+b-a=b$ et $a-b+b=a$
- l'opposé est involutif

Comparaison

- La comparaison signée de deux entiers 32 bits est équivalente à la comparaison dans Z de leurs représentations signées. On en déduit les propriétés d'ordre (réflexivité, antisymétrie, transitivité)
- Si $a<b$, alors $a<a+1\leq b\geq b-1$ (mais on n'a pas a≤b+1)
- min_signed (resp. max_signed) est minimal (resp. maximal), pour l'ordre sur les représentations signées
- on a $a<a+1$ sauf si a=repr max_signed
- de même, on a $a-1<a$ sauf si a=repr min_signed
- si $a\leq a+b$ et si $b\geq 0$, alors la représentation signée de $a+b$ est égale à la somme des représentations signées. Ce théorème est très important car il permet de prouver que si $c\geq b\geq 0$ alors $0\leq c-b\leq c$ et si $a\leq a+c$, alors $a\leq a+b\leq a+c$.

On définit alors la notion d'intervalles disjoints : on dit que $[a,b]$ et $[c,d]$ sont disjoints si, et seulement si, il n'existe aucun entier x tel que $a\leq x\leq b$ et $c\leq x\leq d$. Cette notion est très importante car elle permet d'affirmer qu'une écriture mémoire d'un chunk de k octets à un

offset o d'un bloc n'influe pas sur la lecture mémoire d'un chunk de k' octets à un offset o' du même bloc, dès lors que les intervalles $[o, o+k-1]$ et $[o', o'+k'-1]$ sont disjoints. On a alors la propriété que si b<c, alors $[a, b]$ et $[c, d]$ sont disjoints.

Mais la part la plus importante de cette librairie concerne la congruence modulo 4. En effet, comme on manipule des chunks de 4 octets, on est amené à énoncer des propriétés sur des offsets. Or, on considère les offsets « de quatre en quatre ». En fait, ils sont congrus entre eux modulo 4.

On définit la congruence modulo 4 comme dans Z, mais en utilisant les opérations des entiers 32 bits : deux entiers 32 bits a et b sont congrus modulo 4 si et seulement si $b - a = 4k$ avec la soustraction et la multiplication 32 bits. On dit qu'un entier est divisible par 4 si et seulement si il est congru à 0. Alors, on a les propriétés suivantes :

- la congruence modulo 4 est une relation d'équivalence (réflexive, symétrique, transitive)
- la congruence modulo 4 est compatible avec les opérations d'opposé, d'addition et de soustraction 32 bits
- si $a \equiv b$, et q est divisible par 4, alors $a - q \equiv b$ et $a \equiv b + q$.
- si a et b sont congrus modulo 4 et tels que $a \leq b$, alors, soit $a = b$, soit on a $a + 4 \leq b$ et $a \leq b - 4$. Cette propriété est très importante pour tous les raisonnements d'ordre entre les offsets.

A.2 Listes

Les listes utilisées sont celles de Coq, définies dans la librairie standard List, similairement à celles d'OCaml. Des fonctions sur les listes y sont déjà proposées, comme la concaténation ++. Cependant, il faut définir des prédicats supplémentaires.

Appartenance à la liste Le fait d'appartenir à la liste est défini par le prédicat inductif member : soit l'élément est dans la tête, soit il appartient à la queue.

```
Inductive member (e : A) : list A → Prop :=
| member_head (l : list A) : member e (cons e l)
| member_tail (l : list A) (_ : member e l) (a : A) : member e (cons a l)
.
```

FIG. 59 – Appartenance d'un élément à une liste (member)

Ce prédicat est utilisé partout dans les sources de mon stage, que ce soit pour la modélisation de la mémoire ou la preuve de l'étape de marquage.

Ce prédicat peut être affiné en rajoutant en paramètre la position de l'élément dans la liste. On obtient alors le prédicat inductif find_in_list, suivant la structure à la fois de la liste et de l'entier naturel indiquant la position de l'élément dans la liste.

```
Inductive find_in_list : list A → nat → A → Prop :=
| find_in_list_O : ∀ a l, find_in_list (a : :l) O a
| find_in_list_S : ∀ l n r, find_in_list l n r → ∀ a, find_in_list (a : :l) (S n) r
.
```

FIG. 60 – Recherche d'un élément dans la liste par sa position (find_in_list)

Ce prédicat est utilisé dans la réalisation abstraite d'un chemin, étant donné les listes des objets fils pour chaque objet du tas, ainsi que les listes de listes de racines, le chemin indiquant le numéro de l'objet à chercher dans ces listes.

Suffixe La notion de suffixe d'une liste est définie par un prédicat inductif, is_suffix_of :

- une liste est suffixe d'elle-même
- si une liste est suffixe d'une liste m, alors elle est suffixe de toute liste $a :: m$

```
Inductive is_suffix_of : list A → list A → Prop :=
| is_suffix_of_id : ∀ l, is_suffix_of l l
| is_suffix_of_cons : ∀ l m, is_suffix_of l m → ∀ a, is_suffix_of l (a::m)
.
```

FIG. 61 – Suffixe (is_suffix_of)

Liste sans doublons Une liste est sans doublons, si et seulement si :

- soit elle est vide
- soit sa queue est sans doublons et sa tête n'appartient pas à sa queue

```
Inductive no_duplicates : list A → Prop :=
| no_duplicates_nil : no_duplicates nil
| no_duplicates_cons : ∀ a l, (member a l → False) → no_duplicates l → no_duplicates (cons a l)
.
```

FIG. 62 – Liste sans doublons (no_duplicates)

Ce prédicat inductif est utilisé dans la preuve de l'algorithme abstrait de marquage, où on doit montrer qu'un objet n'est pas dépilé du cache plusieurs fois. Il est également utilisé dans la preuve du fait que deux objets distincts du tas soient disjoints, comme conséquence du fait que cette liste soit triée suivant un ordre irréflexif.

Concaténation La concaténation de listes ++ est prédéfinie dans la librairie standard List de Coq. Il faut alors montrer un certain nombre de théorèmes pour l'utiliser conjointement avec les prédicats définis dans la librairie supplémentaire List_addons.

- Si l=l0++l ou l=l++l0, alors l0 est vide.
- ++ est régulière à gauche et à droite (l++l1=l++l2 peut être simplifiée en l1=l2, et la même chose avec l à droite)
- e est un élément de g++d si et seulement si e est un élément de g ou de d
- Si l1++l2 est sans doublons, alors un même élément ne peut pas appartenir à la fois à l1 et à l2. C'est la propriété intuitive qu'on attend d'une liste sans doublons.
- Si b est élément d'une liste l, alors l peut être décomposée en l1++b : :l2.

On définit alors l'aplatissement d'une liste de listes, sous la forme d'une fonction.

Cette fonction est utilisée pour exprimer qu'un objet est une racine : s'il appartient à la liste obtenue par aplatissement de la liste des listes des racines.

Alors, un élément x est dans une liste aplatie flatten l si et seulement si il existe une liste m appartenant à l et telle que x soit élément de m.

```
Fixpoint flatten l :=
match l with
| nil ⇒ nil (A := A)
| a : :b ⇒ a ++ flatten b
end.
```

FIG. 63 – Aplatissement d'une liste (flatten)

A.3 Boucles

A.3.1 Boucles à corps dans Set

Une boucle est définie à partir d'un type A quelconque représentant un état abstrait, et d'un *corps*, qui prend en paramètre un état de ce type A, et renvoie un couple formé de :

- un booléen indiquant s'il faut sortir de la boucle ou en exécuter une nouvelle itération
- l'état abstrait après une itération de la boucle.

Alors, on dit que l'état *a'* est obtenu à la suite d'une boucle de corps *body* à partir de l'état *a* si, et seulement si :

- le corps *body a* renvoie le booléen false (sortie de la boucle) et l'état *a'*
- ou alors, le corps *body a* renvoie le booléen true (demande d'une nouvelle itération) et un état *a"* tel que *a'* est obtenu à la suite d'une boucle à partir de l'état *a"*

C'est un prédicat inductif, loop.

```
Inductive loop_prop : A → A → Prop :=
| loop_prop_interrupt : ∀ a a',
body a = (false, a') → loop_prop a a'
| loop_prop_continue : ∀ a a' a",
body a = (true, a') → loop_prop a' a" → loop_prop a a".
```

FIG. 64 – Boucle à corps dans Prop (loop_prop)

Les deux cas du prédicat étant disjoints, la boucle est fonctionnelle : si elle termine, alors son résultat est unique.

Lorsqu'on a une boucle, on est intéressé par sa terminaison. Il est facile de voir que la terminaison dans Prop (c'est-à-dire, `exists a', loop_prop a a'`) est équivalente à un prédicat inductif loop_term obtenu à partir de loop_prop en effaçant le second argument résultat de l'exécution de la boucle.

```
Inductive loop_term : A → Prop :=
| loop_term_interrupt : ∀ a a',
body a = (false, a') → loop_term a
| loop_term_continue : ∀ a a',
body a = (true, a') → loop_term a' → loop_term a.
```

FIG. 65 – Prédicat de terminaison de la boucle (loop_term)

Il reste donc à savoir si, étant donné une preuve du prédicat de terminaison, on peut construire dans Set le résultat de l'exécution de la boucle, afin d'obtenir le théorème suivant :

Theorem loop (A : **Set**) ($body$: $A \to$ bool $\times$ A) :
$\forall$ k : A, loop_term (A := A) $body$ $k \to$
{ k' : A | loop_prop (A := A) $body$ k k' }..

Ce théorème est montré en utilisant la tactique refine :

```
refine (
  fix f a (h : loop_term a) :=
  let k := body a in
  (match k as k0 return k = k0 -> _
   with     | (true, a') => fun h0 =>
     match f a' _ with
     | _ => _ (* cas de nouvelle itération *)
     end
   | (false, a') => fun h0 => exist _ a' _  (* cas d'interruption *)
  end) (refl_equal _)) .
```

Cette tactique permet d'effectuer un raisonnement par induction sur la preuve du prédicat de terminaison, en s'assurant que, dans le cas où une nouvelle itération est requise, l'appel récursif à f est effectué sur la preuve de terminaison de la boucle à partir de l'état après la première itération.

Un tel raisonnement par la tactique refine a été essayé au niveau de la preuve d'existence de la liste des objets du tas (block_description_from_to). Cependant, au moment de valider la preuve, Coq se mettait à boucler, probablement au niveau du type-checking. C'est pourquoi il a fallu abstraire et généraliser le concept de boucle. D'où le développement de cette librairie.

A.3.2 Boucles à corps dans Prop

Le problème avec les boucles à corps dans Set est qu'il est difficile de les imbriquer, car la boucle elle-même reste dans Prop, ou au mieux pour l'obtenir dans Set il faut calculer une preuve de terminaison.

Il faut donc définir le concept de boucle *générique* à corps dans Prop, et non plus dans Set.

Dans ce cas, le corps de la boucle n'est plus une fonction, mais un prédicat à trois arguments :

- l'état initial
- un booléen indiquant s'il faut poursuivre l'exécution de la boucle, ou s'il faut l'arrêter
- l'état après une itération

Alors, le prédicat genloop_prop se définit de manière analogue à loop_prop, suivant les deux cas, en fonction du booléen fourni en paramètre du corps de la boucle.

Inductive genloop_prop : A $\to$ A $\to$ Prop :=
| genloop_prop_interrupt : $\forall$ a a',
p a false a' $\to$genloop_prop a a'
| genloop_prop_continue : $\forall$ a a' a",
p a true a' $\to$ genloop_prop a' a" $\to$ genloop_prop a a".

FIG. 66 – Boucle générique (genloop_prop)

A priori, les cas ne sont pas disjoints, car le corps de la boucle est un prédicat et non une fonction. Cependant, si ce prédicat est supposé fonctionnel (c'est-à-dire que, étant donné un

état initial, au plus un booléen et un état après une itération sont en relation avec lui par le corps de la boucle), alors la boucle est fonctionnelle.

De la même façon qu'avec la boucle à corps dans Set, le prédicat de terminaison s'écrit en effaçant l'état final dans le prédicat genloop_prop, obtenant alors le prédicat inductif genloop_term. Et alors, la terminaison dans Prop est équivalente à ce prédicat de terminaison.

```
Inductive genloop_term : A → Prop :=
| genloop_term_interrupt : ∀ a a',
p a false a' →genloop_term a
| genloop_term_continue : ∀ a a',
p a true a' → genloop_term a' → genloop_term a.
```

FIG. 67 – Prédicat de terminaison de la boucle générique (genloop_term)

Il reste alors le problème de la construction dans Set de l'état final, sous hypothèse de terminaison. Pour cela, il faut deux hypothèses sur le corps de la boucle :

- le corps de la boucle doit être fonctionnel
- le corps de la boucle doit être *décidable* : si pour un état initial m on peut trouver dans Prop un booléen b et un état m' après une itération, alors b et m peuvent être construits dans Set.

Alors, sous ces deux hypothèses ainsi que la terminaison de la boucle, on peut montrer, de façon analogue à loop_prop, que l'état final peut être construit dans Set.

```
Definition genloop : ∀ a, genloop_term a →
{a' | genloop_prop a a'}.
```

On remarque que dans les deux cas, la boucle est fonctionnelle et décidable (c'est-à-dire que si l'état final peut être construit dans Prop, alors il peut être construit dans Set). Alors, grâce à genloop, on peut imbriquer deux boucles si la boucle englobante est une boucle générique. En effet, la fonctionnalité et la décidabilité de la boucle imbriquée servent à prouver celles du corps où cette boucle est logée.

Références

[ABN+] Andrew Appel, Sandrine Blazy, Francesco Zappa Nardelli, Aquinas Hobor, and Adriana Compagnoni. Concurrent c minor. `http://www.cs.princeton.edu/~appel/cminor`.

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7, 1999.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions.* Springer Verlag, 2004.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, number 4085 in Lecture Notes in Computer Science, pages 460–475. Springer Verlag, 2006.

[BL05] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for c-like imperative languages. In Springer-Verlag, editor, *International Conference on Formal Engineering Methods*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer Verlag 2, 2005.

[CGN03] Solange Coupet-Grimal and Catherine Nouvet. Formal verification of an incremental garbage collector. *The Journal of Logic and Computation*, 16(4) :352–373, 11 2003.

[Coq] The coq proof assistant. `http://coq.inria.fr`.

[ea07] Xavier Leroy et al. The compcert certified compiler back-end. `http://pauillac.inria.fr/~xleroy/compcert-backend`, 2007.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd Symposium Principles of Programming Languages*, pages 54–68. Springer Verlag, 2006.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris 7, 2004.

[MSLL07] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *PLDI '07 : Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 468–479, New York, NY, USA, 2007. ACM Press.

[Ram07] Tahina Ramananandro. Sources coq de la preuve du gc mark and sweep. `http://www.eleves.ens.fr/~ramanana/travail/gc`, 2007.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`, 1992.

Table des figures