# Notes de lecture: Automatic Space Bound Analysis for Functional Programs with Garbage Collection

Hector Suzanne

## Introduction

Les techniques actuelles d'analyse de coût, basées sur

- L'interprétation abstraite,
- la résolution de récurrence,
- les systèmes de types,
- la logique de programme,
- les assistants de preuves, ou
- la réécriture

supposent que la mémoire est gérée manuellement, ou ignorent simplement la libération mémoire ! Il existe par contre des exceptions: Albert (2009), Braderman (2008), Unnikrishnan (2003, 2009), qui fonctionnent de manière similaire: (1) rendre le déclenchement du ramasse-miette (ou *GC*) explicite, (2) extraire et résoudre les récurrences ainsi générées. Notre méthode, basée sur le typage est naturellement composable et les dérivations peuvent servir de *certificats*.

Nous modélisons l'utilisation mémoire "à marée haute" (c.à.d. maximale) quand le programme est muni d'un GC idéal qui libère la mémoire dès qu'elle devient hors d'atteinte. Ça a quand-même des usages pratiques: quand l'usage mémoire dépasse la borne théorique, on sait que le GC peux récupérer de la mémoire, on peut diminuer la pression mémoire en donnant plus que la borne, etc.

Nous trois contributions techniques sont:

- Une sémantique de coût opérationnelle pour un langage fonctionnel type ML avec GC idéal, qui tient à jour une *free list* de position nommées, intéressante en soit.
- Un système de type basé sur AARA, dérivant des bornes des coûts mémoires en présence de GC idéal. Notre système se base sur le fait que la libération mémoire se ramène au filtrage par motif destructif tant que l'inspecté est une structure linéaire. On étend ce principe au programmes non-linéaires, en prenant en compte le partage.

- Nous prouvons la cohérence de notre AARA par rapport à la sémantique de coût avec GC idéal. On procède en liant le tas de la sémantique avec GC à des partitions de tas dans une sémantique sans partage (qui copie les données) et sans GC. On borne ensuite le coût dans la première sémantique par le coût dans la seconde. Notre implémentation au sein de RaML couvre la majeur partie du langage cible de l'outil, mais notre preuve ne couvre qu'un langage de premier ordre avec listes et tuples. Ainsi, pas besoin d'allouer (ou de copier) des fermetures.

# Sémantique de coût avec GC

## Préliminaires

On travaille sur langage du premier ordre avec booléens, entiers naturels, tuples et listes comme structures; et let non-récursifs, conditionelle, application (au premier ordre), filtrage sur paire et listes, et partage explicite comme termes. Le partage explicite introduit deux notions distinctes de *linéarité*: un programme est *syntaxiquement linéaire* si chaque variable est utilisée une fois, et *sémantiquement linéaire* si chaque adresse du tas est utilisé une fois.

Les programmes sont en forme *-liftés*, toutes les fonctions sont définies à la racine du programmes et mutuellement récursives par défaut. Un programme $P$ a une signature $\Sigma : Var \to fun(t_1, t_2)$ qui le spécifie, que l'on note $P : \Sigma$. À l'exécution, le programme est muni d'un contexte d'évaluation $V : Var \to Val$ et d'un tas $H : Loc \to Val$.

Dans un tas $H$ , une adresse $l \in Loc$ est atteignable (ou-non) depuis une valeur $v$. On note $L = reach_H(v)$ le multi-ensemble des adresses atteignables transitivement depuis une valeur $v$. C'est un multi-ensemble car on considère le partage dans notre preuve de cohérence, mais l'approximation par un ensemble simple est mentalement utile. On étend point-par-point cette définition à $reach_H(V)$, et à $locs_{V,H}(e)$ les adresses accessibles depuis les variables libres d'une expression $e$.

## $\mathcal{E}_{GC}$, la sémantique de coûts avec GC

Les jugements sémantiques de $\mathcal{E}_{GC}$ ont la forme

$$V, H, R, F \vdash_{P:\Sigma} e \Downarrow v, H', F'$$

et signifie: "Dans le programme $P$ de signature $\Sigma$, avec en environnement $V$ et un tas $H$, une *continuation mémoire* $R$ et une *free-list* $F$, l'expression $e$ s'évalue en une valeur $v$ dans un tas $H'$ est une *free-list* $F'$".

Le quadruplet $(V, H, R, F)$, abrégé en $C$, contient une continuation mémoire $R$ qui est un multi-ensemble d'adresses dénotant l'ensemble des adresses nécessaires à l'évaluation de la suite du programme après $e$. Ces adresses ne peuvent

donc pas êtres libérés durant l'évaluation de $e$. $F$ est un ensemble d'adresses dont le contenu à été libéré et peuvent être ré-utilisées. Un état, ou *calcul* est alors caractérisé, et identifié à un couple $(C, e)$.

Par rapport à un ensemble d'adresses racines $L$ d'un calcul, défini comme l'union des adresses de $R$ et de celle présentes dans $e$, les *miettes* sont définies comme $collect(R, L, H, F, e) = \{l \in H | l \notin F \cup L \cup R\}$.

On définit enfin les *configurations bien-formée* comme les couples $((V, H, R, F), e)$ satisfaisant les conditions suivantes:

**allocation automatique** $dom(H) \subseteq reach_H(V) \cup R \cup F$. Le tas ne contient que ce qu'on y a mis.

***free-list* libre** $reach_H(V) \cup R \subseteq H - F$. La mémoire atteignable est dans le tas mais pas la *free-list*.

**GC parfait** $collect(R, reach_H(V), H, F) = \emptyset$. Toute la mémoire est libérée immédiatement pendant les transitions, jamais entre.

# AARA linéaire avec GC

## AARA linéaire depuis AARA classique

*AARA* linéaire est une des premières versions de AARA sur laquelle a travaillé Hoffmann. On peut l'inclure dans la version complète, basée sur les polynômes multivariables, en réduisant l'ensemble des polynômes de bases de chaque type: L'ensemble des indices dans AARA linéaire ici est, pour les listes, $I = \{\star, I_{\texttt{nil}}, I_{\texttt{cons}}\}$, les fonctions associés ayant comme co-domaine $\{1\}$ ou $\{0, 1\}$.

Avec AARA classique, on obtient des bornes de la forme

$$\texttt{append} : L^p(a) \times L^q(a) \overset{r/r'}{\to} L^s(a) \wedge p \geq s + 1 \wedge q \geq s \wedge r \geq r'$$

qui signifie que, le potentiel *par élément* des argument est $p$ et $q$ et qu'il faut allouer une cellule par élément du résultat ($p \geq s + 1$). Si l'argument initial peut être récupéré par le GC, `append` a en réalité un coût mémoire par élément nul, que AARA ne prend pas en compte.

Les jugements de typage sont eux aussi simpifiés: on a

$$\Sigma; \Gamma \vdash^q_{q'} e : A$$

pour, en AARA normale,

$$\Sigma; \Gamma; (Q, q_\star = q) \vdash e : (A, (Q, q_\star = q'))$$

que l'on a déduit des Théorèmes de cohérences pour les deux AARA avec pour ainsi dire des langages et métriques identiques.

### AARA avec GC

Avant analyse, les programmes sont mis en forme *let-share-normale*, qui est syntaxiquement linéaire. La gestion du GC consiste à, pour chaque type, associer une perte de potentiel constantes aux valeurs quand elles sont partagées, avec un gain de potentiel complémentaire quand elles sont inspectées par un filtrage, qui est nécessairement destructif dans un langage syntaxiquement linéaire. L'opération de partage doit donc être augmentée est customisable par métrique de coût, tout comme le filtrage.

Avec comme exemple un programme ou l'on trouve un appel à `append` que le GC rend gratuit, on peut alors rendre compte du potentiel gagné en analysant l'appel séparement. On obtient alors, pour cet appel:

$$\texttt{append} : L^p(a) \times L^q(a) \overset{r/r'}{\to} L^s(a) \wedge p \geq s \wedge q \geq s \wedge r \geq r'$$

La suppression de la constante 1 indique que les cellules `cons` utilisées en tête de liste concaténée sont réutilisées car la liste argument a été libérée.

## Cohérence

On prouve que, si notre système de type indique qu'une évaluation termine dans un contexte et avec un potentiel donné, et que selon la sémantique classique nommé "oper" (qui ignore la gestion mémoire), cette évaluation donne lieu à une valeur, alors en prenant en compte le GC, une free-list de taille suffisante pour le contexte et le potentiel de départ suffit à cette évaluation et renvoie la même valeur.

Plus formellement, il nous faut définir une équivalence de contexte, et une de valeur, entre les deux sémantiques, et raisonner sur cette équivalence. Le gros de la preuve est de montrer que l'évaluation dans la sémantique linéaire à copie consomme au moins autant de potentiel que dans celle avec GC. Il suffit en suite de montrer la cohérence avec cette nouvelle sémantique plus simple.

Le théorème final est:

Si $\Sigma; \Gamma \vdash^{q}_{q'} e : A$ avec

- $H \Vdash V : \Gamma$, et
- $V, H \vdash_{oper} e \Downarrow v, H'$

et si $C = (W, Y, F, R)$ avec

- $V, H \equiv W, Y$, et
- $|F| \geq \Phi_{V,H}(\Gamma) + q$,

alors:

- $C \vdash_{gc} e \Downarrow w, Y', F'$, et $v \equiv^{Y'}_{H} w$

# Évaluation et perspectives

Nous avons implémenté cette nouvelle analyse dans une version de *RAML*, en prenant en argument la taille initiale de la free-list et en séparant les cas où l'analyse garantie que le dépassement mémoire. On prends en compte une stratégie d'allocation ou toutes les données de taille statique sont allouées sur la pile. Seuls les constructeurs de types sont alloués sur le tas. Notre implémentation supporte les types utilisateurs.

On note une différence importante avec la version présentée ici: dans ce papier, le constructeur `nil` est de coût 0, car il n'a aucun champ. Pour prendre en compte des ADT utilisateurs qui peuvent avoir plusieurs constructeurs d'arité 0, on les alloue sur le tas et donc leur donne un coût non-nul. Enfin, les fermetures en racine du programme doivent être allouées, mais le sont sur une free-list différente. En général, notre version autorise la création de fermeture locale sans les prendre leur libération en compte dans le calcul de taille.

Notre évaluation montre que la prise en compte du GC est clé pour l'obtention de borne optimales pour certains algorithmes en place, comme *quicksort*, *selection sort*, et des recherches dans des arbres. Des bornes asymptotiquement optimale ont étés atteinte pour la création de produit carthésiens (`'a list -> 'b list -> ('a * 'b) list*`), là où sans GC, la borne est faussée d'un facteur 2.

Dans des travaux futurs, il serait intéressant de prendre en compte l'allocation *statique* des constantes de programmes, qui sont une optimisation courante. Une piste est d'utiliser une seconde passe d'analyse où ces valeurs constantes on un coût nul, et de la composer avec la notre pour prendre en compte séparement le coût propre des manipulation des calculs et les allocations de constantes dynamiques. Aussi, la prise en compte de la libération des fermetures serait une amélioration bénéfique. On pourrait, par exemple, libérer les fermetures lors de leur applications et traiter leur partage uniformément du reste des valeurs. Malheureusement, la taille des fermetures est difficile à estimer statiquement et elle peuvent contenir leur propre potentiel, ce que RAML ne supporte pas pour le moment.