Memory Consumption Analysis for a Functional and Imperative Language

Hector Suzanne

Introduction

On propose un langage à la ML avec gestion manuelle de la mémoire par région et un système de type et d'effet. On muni ce langage d'une analyse de consommation mémoire et de détection des erreurs de gestion des régions. Notre analyse distingue grâce au système d'effet les fonctions *pures* et *impure* du point de vue de la mémoire et les traite différemment: les fonctions pures sont traitées par AARA linéaires et les impures par invariants de boucles sur les régions, inférés ou annotés dans le code source.

Dans les deux cas, les analyses permettent de trouver des bornes symbolique de l'occupation mémoire des fonctions, qui peuvent être combinées, en prenant soin des effets de bord, pour trouver une borne totale sur le programme. Notre contribution consiste en un carde formel permettant de faire interagir les résultats de différentes analyses de coût mémoires, choisies en fonction du *paradigme de programmmation* utilisé.

/Le lecteur passe la seconde section, redondante vu les notes qu'il a déjà sur les travaux portant sur l'analyse statique de gestion mémoire. On note l'existence de "Safe" qui utilise l'interprétation abstraite et "JConsume" qui utilise les invariants de boucles./

Langage et analyse

Survol

On veut implémenter un langage de programmation moderne (*un système F, donc*) sans ramasse-miette, car le comportement hautement dynamique de se dernier est peu prévisible. Dans le passé, Tofte & Talpin on implémentés le -calcul typé avec *call-by-value* avec une pile de région comme modèle mémoire, puis Fluet & al on étendu cette approche à une interface monadique de la gestion de région, ne nécessitant pas de discipline de pile pour les régions. En introduisant un modèle de coût pour les cellules allouées sur le tas, on veut quantifier l'occupation mémoire d'un programme donné.

L'analyse que nous développons pour cela procède fonction par fonction, en séparant les fonctions *pures*, codées en style fonctionnel, et les fonctions *impures*, codées en style impératif. Notre système d'effet détecte si une évaluation peut éventuellement écrire dans une région ρ. Si c'est le cas, elle est traitées comme impure, sinon, elle est considérée comme pure. Cela n'exclue pas la mutabilité, mais impose qu'elle soit invisible de l'extérieur de la fonction.

Syntaxe et typage

On utilise une syntaxe classique, avec comme différences:

- Les fermetures, paires, listes doivent être annotées $@\rho$ signifiant qu'elles sont allouées dans une région ρ précise.
- Les types de ces structures sont annotés de même avec la région où ils existent.
- On définie trois primitives pour les régions: `newrgn`, `aliasrgn` et `freergn`.
- On définit des *témoins de région*, de type $hnd\rho$ indicé par les régions. On peut quantifier sur ces indices, mais jamais au dessus d'un quantificateur de type.

Au dessus de cette syntaxe, un définit un système de type et d'effets qui garanti la discipline de pile nécessaires à la bonne gestion mémoire et permet d'approximer statiquement la topologie du tas. Les jugements sont de la forme

$$C; \Gamma \vdash e : \tau,\ \Gamma'; C'; \varphi$$

où φ est un ensemble de *d'effets* de la forme $read(r)$, $write(r)$ ou $alloc(r)$, et C est un ensemble de *capacités* de la forme r^1 ou r^+ signifiant le mode linéaire ou illimité d'accès à une région. Les types des fonctions sont eux annotés d'effets produits par l'évaluation, de capacités nécessaires à celle-ci, et de capacité disponibles après.

Le typage garanti que les régions sont linéaires quand elles sont crées avec `newrgn` , et qu'elles le sont redevenues quand elles sont détruites par `freergn` . Entre temps, la primitive `aliasrgn in e` donne le mode d'accès illimité à ρ dans e.

Le lecteur note que dans la règle d'évaluation des appels de fonction, l'argument est évalué avec l'environnement et les capacités post-jugement de la fonction, semblant indiqué une évaluation ***call-by-name****. Erreur?*

Analyse des fonctions

Pour les fonctions pure, ont utilise une AARA linéaire, mais région par région, pour pouvoir considérer le potentiel gagné par la libération. On pensera à compter le coût en taille des fermetures à leur création (*où?, dans quelle règle?*), et à permettre le partage de valeur et de potentiel associé, nécessairement au sein de la même région (*avec un prédicat* `share` *pas définit*).

Pour les fonctions impures, on infère des invariant de boucles linéaires (*probablement par interprétation abstraite*), ou on utilise ceux fournit en annotation. Même si une fonction peut être analysée comme impure, elle peut être *opérationellement pure*, ce qui peut simplifier l'analyse de ses appelants.

Les résultats de ces analyses ne prennent pas en compte les effets de bords, qu'il faut gérer séparément pour propager les annotations de taille en garantissant la sûreté. Les seuls effets du langage sont l'écriture dans les références, on décide dont d'annoter le type `ref` avec d'une part la région dans laquelle la référence vie, et d'autre part celle dans laquelle la valeur pointée vie (*En Rust, les références sont génériques en leur durée de vie et celle du pointé*).

Hélas, cela ne suffit pas à garantir le potentiel mémoire des références, on prend en exemple le code *OCaml* pur suivant:

```
let update x = append x [3;4;5;6;7;8;9;10] in
let r = ref [1;2] in
r := update r
```

la complexité mémoire de la référence à la fin de l'extrait n'est pas déductible de celle de `r`, et dans des cas plus complexe, même pas de celle de `update`. Si les régions sont employés habilement, on peut néanmoins connaître les tailles des structures référencées. On munit notre analyse d'un opérateur `size` à cet effet, qui doit être fournit par l'utilisateur quand l'inférence échoue.

Le lecteur note la présence d'un opérateur `update` *du typage qui n'est défini que par un exemple, est qu'il n'a pas compris. Aussi, la règle d'analyse des conditionelles ne prend pas en compte les allocations éventuelles du booléen inspecté.*

Conclusion

Lire *Linear Regions Are All You Need* de Fluet, Morrisett & Ahmed.