Notes de Lecture: Arrays and References in Resource Aware ML

Hector Suzanne

Introduction

Les méthodes d'analyses statiques classiques permettent d'inférer des bornes impressionnante sur les programmes impératifs avec données dans . Mais la mutabilité, les dépendances cycliques et la faiblesse relative du typage met un frein au bornage des structures de données basées sur les pointeurs. Issu de la programmation fonctionnelle pure, l'analyse automatique de ressource par amortissement (AARA) permet de prendre parti du typage statique fort pour inférer sans interaction utilisateurs des bornes sur la tailles des structures de données fréquentes en programmation de haut niveau.

Néanmoins, les méthodes basées sur AARA, notamment l'outil RAML, n'infère pas de bornes sur les tailles des données stockés à travers des références ou des tableaux, ce qui est un problème pour la programmation impérative. Notre travail étend RAML pour inférer ce type de bornes et fonctionne sur des ressources *monotones* (c.à.d. pouvant être libérables).

L'avantage de notre méthode est d'être pleinement compatible avec l'inférence de coût à base de programmation linéaire de RAML, qui est plus simple que le plupart des système similaire pour la programmation impérative, basée, par exemple sur la logique de séparation. Aussi, nous reprenons le design du système de type ML, qui ne dénote pas les effets associés aux valeurs, mais garantie le type de la valeur, au travers des références et des tableaux. Même si notre analyse ne produit ses meilleurs résultats que pour un certain idiome impératif, le suivi de cette discipline de programmation permet de garantir des bornes mémoires sur des algorithmes de recherche et mutation de graphe cycliques.

Voir "Alias Typing" de l'ESOP'00

Langage considéré et son analyse statique

Langage

On utilise un λ-calcul avec ADT, en forme *let-share-normal*, avec un système de type affine: chaque variable est utilisée au plus une fois. On muni ce langage

de primitives pour les références et les tableaux:

$$
\begin{aligned}
e \ ::= & \ \text{ref}\ x \mid !x \mid x_1 := x_2 \mid \text{swap}(x_1, x_2) \\
& \ \text{create}(x, e) \mid \text{get}(x_1, x_2) \mid \text{set}(x_1, x_2, x_3) \mid \text{aswap}(x_1, x_2, x_3) \mid \text{lenght}(x)
\end{aligned}
$$

Les valeurs sont les classiques, plus des pointeurs, soit nuls, soit des adresse mémoires valides $l \in Loc$, et des tableaux (σ, n), où σ est une fonction de $\{0, ..., n-1\}$ dans Loc.

Avec la sémantique opérationnelle à coût et le système de type usuel, on prouve un théorème de cohérence qui garantie qu'un programme bien typé sous un environnement bien-formé termine avec un coût ou diverge.

Analyse de ressource automatisée

Les types annotés en AARA linéaire pour notre langage sont:

$$A \ ::= \ \text{unit} \mid B \mid X \mid \eta X^Q.(C_i : A_i)_i \mid A \ \text{ref} \mid A \ \text{array}^q$$

Seuls les tableaux et les ADT sont alloués sur le tas, et donc annotés d'un coût, q, qui est fixe par élément pour les tableaux, et Q, qui associe récursivement un coût à chaque constructeur de chaque ADT. On définit les entiers unaires N^q, qui serviront d'indices dans les tableaux (l'entier n ayant le coût $q_N \cdot n$), et les listes $L^q(A)$ (de coût q par `cons`).

Le potentiel linéaire est défini normalement, le potentiel d'un pointeur l étant celui de la valeur pointée. Pour modéliser le partage explicite dans `share` via les références, on définit une relation de partage, que l'on note $\text{share}(A \Rightarrow A_1, A_2)$ définie quand $|A| = |A_1| = |A_2|$ et quand on peut partager le potentiel d'une valeur de A entre deux valeurs du même type sous-jacent portant des annotations plus faible, A_1 et A_2. Les références sont systématiquement partageables:$\forall \alpha, \text{share}(\alpha \ \text{ref} \Rightarrow \alpha \ \text{ref}, \alpha \ \text{ref})$. Autrement, elle est définit point-par-point.

Le reste de l'AARA linéaire est classique, voir le cours de Hoffmann.

Potentiel des références et des tableaux

On commence par un exemple édifiant, en pseudo-OCaml:

```
g l = let r = ref l in
      share r as (r1,r2) in
      g' r1;
      append !r2 []
```

Dans notre système de type, on aurait `g'` : $L^q(T)$ ref $\to$ unit. Donc, il est impossible de savoir combien de potentiel il nous reste quand on entreprend d'`append` la liste via la référence partagée… Nous adressons ce problème de la manière la plus simple: en interdisant les variations de potentiels au travers des références. Ainsi, `g'` ne peut pas consommer le potentiel de `l`. Dans une autre approche, où l'on pourrait atteindre le potentiel, il faudrait tracer l'aliasing des référence à travers le programme, ce qui engendre une explosion combinatoire, et donc ne passe pas à l'échelle.

La primitive `swap` permet l'accès au contenu des références en maintenant le potentiel constant. Elle est simplement implémentée par:

```
swap r x = let y = !r in r := x; y
```

Notre système de type garantie alors que `x` a autant de potentiel que la valeur stockée dans `r`. On peut aussi utiliser l'opérateur `!` usuel, mais uniquement quand le potentiel au travers de la référence est inféré comme nul. Cela est réalisé en imposant share($A \Rightarrow A, A$) quand on veut déréférencer depuis une A ref. L'usage de `swap` place le programmeur dans une situation plus délicate, où il est nécessaire de fournir une valeur *par défaut* pour stocker le potentiel, mais cela reste gérable dans la plupart des cas. (*sic.*)

Le traitement des tableaux est rigoureusement identique, via l'usage de `aswap` est de la même condition de nullité de potentiel lors de la lecture dans le tableau.

Preuve de cohérence

La preuve de cohérence possède une unique difficulté: il faut s'assurer que les environnement sur lesquels on compte le potentiel ne comportent pas de références aliassées. Comme les références aliassées sont aliassées ssi elles pointent vers la même adresse, il faut et il suffit de compter le potentiel de ces adresse une unique fois. On l'assure en introduisant un *contexte de mémoire* Δ, qui est une fonction partielle de *Loc* vers les types et un opérateur de "dé-partage" $\otimes$, qui fusionne deux Δ concordant. En construisant le long des termes les Δ nécessaires, on défait l'aliasing. Il faut adapter la notion d'environnement bien-fondé pour qu'elle garantisse en plus la cohérence des contextes de mémoire, mais c'est immédiat.

Il suffit en suite de définir point-par-point le potentiel du Δ à la racine, c'est le potentiel des valeurs pointées par les références, et de définir que les références ont un potentiel propre nul. La cohérence est obtenue par les moyens usuels.

Inférence des types annotés

En pratique, nous pouvons inférer les annotations de coûts pour AARA polynomiale est en présence d'ordre supérieur (toujours en comptant uniquement les coûts propres). Mais même dans le cas d'AARA linéaire, il est nécessaire d'adapter les notations de potentiel des fonctions: si elles sont référencés, il

faut adapter l'annotation au site d'appel. Heureusement, AARA polynomiale à une variable associe déjà aux fonctions un *ensemble* d'annotation. Les règles de types restent identiques (*en tout cas, isomorphes*).

Dans AARA polynomiale multi-variable, si un *polynôme de base* d'une valeur sont une référence apparaît dans un produit avec un autre polynôme, l'inférence n'a pas été résolue. On impose alors que le potentiel compté *via* la référence soit nul. Cela revient à négliger l'aliasing. (*Le lecteur note que c'est encore un problème de linéarité…*).

Travaux connexes, Conclusion

Les travaux récents sur RAML permettaient de garantir l'invariance du potentiel par rapport au contenu des références et des tableaux. Notre contribution est la capacité à prendre en compte se potentiel. Des analyses sur l'état mutable sur du tas sont intégrés à des systèmes d'analyse statiques basés sur la logique de séparation et le typage orienté-objet, avec une plus grande expressivité. Mais l'avantage de notre méthode et qu'elle peut inférer automatiquement les bornes, ce qui n'est leurs cas. L'inférence fonctionne même en présence de structures mutables cycliques.

D'autres systèmes existent, sans inférence, notamment basé sur les types dépendants linéaires, mais ils ne gèrent presque pas les effets de bords et ne sont pas, à notre connaissance (*à la mienne non plus*) automatisables. A base de relation de récurrence non-deterministes, les systèmes de *Albert* gèrent les structures mutables au niveaux du bytecode via des registres virtuels, mais nous ne savons pas à quel point cette technique supporte les structures imbriquées, qui demandent de suivi de pointeur.

Contrairement aux travaux semblables précédents, notre AARA enrichi le langage hôte avec des structures *admissibles* au niveau du code qui lui sont bien adapté: si le potentiel du programme dépend de données derrière des références, il faut y accéder via `swap/aswap`.