Notes de Lectures: Towards Automatic Resource Bound Analysis for OCaml

Introduction

On introduit une analyse statique de ressources pour les programmes OCaml *fonctionnels*, *d'ordre supérieur*, *avec polymorphisme* et *types utilisateurs*. On obtient en résultat des polynômes multivariables décrivant l'usage de ressources tel que le temps d'exécution au pire (WCET), l'occupation mémoire, la consommation d'énergie ou encore le nombre d'appel à une API. On utilise un nouveau système de type et une nouvelle sémantique de la machine virtuelle *ZINC* d'OCaml. On peut trouver des bornes non-asymptotiques fortes pour des programmes réels, comme les utilitaires de la bibliothèque standard sur les listes, l'algorithme de Dijkstra, ou un client pour Amazon DynamoDB, avec à chaque fois les métriques pertinentes.

Premier exemple

```
type ('a, 'b) ablist =
| Acons of 'a * ('a, 'b) list
| Bcons of 'b * ('a, 'b) list
| ABNil

let sort_the_as the_arg = match ablist with
| Acons (a, tail) -> Acons (quicksort a, sort_the_as tail)
| Bcons (b, tail) -> Bcons (b          , sort_the_as tail)
| ABNil -> ABNil
```

Sur ce fragment de véritable code OCaml, RAML infère comme métriques pertinentes

- `N`, le nombre de noeuds Acons de `the_arg`,
- `K`, le maximum des tailles des arguments de type `'a` des Acons, et
- `L = N+K` la taille de `the_arg`,

et assigne le coût mémoire `13*K*N^2 + 22*K*N + 13*L + 15*N + 11` à la fonction analysée, en terme d'étape d'évaluation dans la sémantique à grand pas de RAML.

Pour ce faire, on extrait un AST typé du compilateur OCaml, puis on infère de nouveaux types contenant des information de potentiel pour chaque expression. A partir d'une déclaration `t : T`, on obtient une autre définition `t : (T'; <q1…qn>))` ou `T'` est proche de `T` (on y précise si les applications sont partielle ou totales), et les `qi` sont des coefficients permettant de définir un potentiel $\varphi(t) = \Sigma q_i * p_i(t)$. Les polynômes p_i sont définis pour chaque constructeurs du type `T`. Notre sémantique induit des contraintes linéaires entre les potentiels. Elles sont résolues par un solveur de programmation linéaire après un parcours de l'AST qui collecte les contraintes et détermine les objectifs.

Analyse de la programmation fonctionnelle

On a distingué trois principales difficultés à l'analyse de ressource: l'application partielle, les arguments d'ordre supérieurs, et les effets de bords

Fermetures et curryfication

On prend comme exemple `val append : 'a list -> 'a list -> 'a list` de la bibliothèque standard. L'appel `let foo = append xs` termine en temps constant, et ne fait que créer une fermeture. Par contre, l'appel `foo ys` renvoie la liste `xs @ ys` en temps et espace $O(xs)$. Donc, l'usage de ressource de `foo` ne dépend pas de son argument `ys`, mais est linéaire en la taille de la valeur capturée `xs`. Hélas, il n'est pas suffisant de dé-curryfier `append` pour retomber sur un cas plus simple: Un appel à une version dé-curryfiée d'`append` nécessite d'allouer une paire pour les deux arguments, alors que dans le cas normal, on produit une fermeture si l'application est partielle, et on appelle `append` sans allocation si les deux arguments sont fournis.

On résout ce problème en raffinant les types des fonctions sur leurs sites d'appels. Les applications partielles peuvent être saturées par eta-expansion, et on change le type de la fonction de `a1 -> … -> an -> t` en `[a1 … an] -> t`, où l'annotation `[…]` indique la saturation de l'appel. L'analyse peut alors rendre compte de l'usage correct des ressource dans le second cas. Hélas, elle est bien plus complexe si les valeurs capturées dans les fermetures interviennent dans le calcul de complexité. L'analyse est donc limitée aux appels annotés `[…]`. De cet exemple ressort le critère principal de notre analyse:

> *La complexité d'un appel de fonction ne doit dépendre que de la taille de ses arguments.*

On reconnaît la possibilité d'étendre le système pour gérer les appels non-saturés, mais cela aurait un coût important en terme de complexité. On pourrait par exemple étendre le raffinement des types avec des types dépendants.

Gestion de l'ordre supérieur

Dans le cas plus simple où on se limite à des estimations linéaires du coût, il est possible de calculer la consommation de ressource en présence d'arguments d'ordre supérieur en se ramenant à l'ordre un. Mais c'est seulement possible quand les arguments sont eux-mêmes analysables, et on doit entrelacer le typage et la génération de contraintes. On préfère séparer les deux processus.

On peut néanmoins analyser les fonctionelles en l'absence d'information sur leurs arguments: Il suffit de supposer les coûts qu'ils engendrent négligeables. L'analyse n'est alors pas veine: On peut déterminer le coût propre de fonctions comme `map` et `fold`, et l'analyse sert aussi de preuve de terminaison. On peut en effet dire qu'une fonction d'ordre supérieur ayant un coût défini termine si et seulement si les appels à ces arguments terminent tous.

Dans les cas où la terminaison de la fonction dépend de propriétés des arguments d'ordre supérieur, on ne peut pas conclure dans le cas général. Mais on peut reprendre l'analyse dans les sites d'appels où ces arguments sont connus, comme dans le cas suivant:

```
(* Impossible de conclure ici *)
let iter f x =
  let y = f x in
  if y = 0
  then ()
  else iter f y

(* mais ici on peut spécialiser pour une valeur présice de `f`*)
let foo x = iter (fun z -> 0) x
```

Effets de bord

L'analyse vérifie que le coût du programme est indépendant des valeurs stockées dans les états mutables. Dans le cas contraire, on peut peux pas poursuivre. Il a été montré que l'analyse à base de polynômes peut être étendue avec des potentiels pour les états mutables, mais cela est ici hors du champ d'étude, qui est l'analyse sur les programmes fonctionnels. RAML peut néanmoins interagir avec l'état mutable pour affiner ses résultat. On peut par exemple, effectuer une analyse d'aliasing sur les références à des fonctions: Si la référence en question ne peut prendre qu'un nombre fini de valeurs (et qu'on peut le détecter), alors on peut calculer un coût au pire pour les appels à travers la référence. Enfin, on peut étendre l'analyse aux programme lançant des exceptions en annotant le type de l'exception et de ses arguments.

Core RAML, première sémantique

Après simplification de l'AST OCaml, on se ramène à un langage simple, où les termes sont sous forme *let-share-normale*: on a remplacé le plus de sous-expressions possibles par les variables créées par let ou share (voir syntaxe). Pour garantir que le programme transformé rend compte de la consommation du programme en entrée, on peut marquer des constructions comme *gratuites* via une meta-syntaxe, qui sont ignorés dans l'analyse. On présente ici un langage réduit, sans types primitifs comme les tableaux, leurs opérations, ou la conditionelle.

Syntaxe

- Les classiques:
  - constructeurs, pattern matching
  - abstraction, application
  - tuples, références
  - `let`, `let rec` de fonctions, variables
- `fail`, pour modéliser les exceptions
- `tick(q )`, une consommation de ressources
- `share x as (y,z) in …`, le partage de valeurs

Les types sont l'unité, les tuples, les références, les fonctions et les ADT classiques.

Définitions sémantique

Les valeurs Val sont

- l la position d'une valeur sur le tas,
- $(l_1, ..., l_n)$ les tuples de valeur,
- $(\lambda x.e, V)$ les fermetures, avec leurs environnements, et
- Cl les applications de constructeur à une valeur.

$K \in Syn$ sont les noeuds de syntaxe, S est une pile d'arguments, et $M : Syn \times \mathbb{N} \to \mathbb{Q}$ est une *métrique de ressource*. Le paramètre naturel permet, par exemple, de spécifier la taille des tuples.

Le modèle mémoire comporte des variables $x \in Var$ des adresses $l \in Loc$ et les valeurs Val. Les variables indicent un environnement (partiel) $V : Var \to Loc$. Enfin le tas est $H : Loc \to Val$ et comporte un pointeur nul avec $H(null) = null$.

Sémantique à grand pas

Le jugement principal

$$S, V, H \vdash_{\mathrm{M}} e \Downarrow w|(q, q')$$

s'entend par:

- Selon la métrique de ressource M
- dans un environnement V,
- avec une pile S et un tas H,
- l'expression e se réduit en w,
- en induisant un coût (q, q').

En général, on note la consommation $(q, 0)$ par q et $(0, -q)$ par $-q$.

Les réductions w sont soit une valeur de retour et un nouvel état (l, H'), soit un blocage $\bot$, soit une non-terminaison $\bigcirc$.

Les consommation de ressources sont de la forme (q, q') où q est la "marée haute", la quantité maximale de ressource dont on à besoin, et q' est la quantité de ressource disponible après l'évaluation. La consommation est paramétrée par la *métrique de ressources* M, qui est une fonction qui à un noeud de la syntaxe k et sa taille n associe le coût propre de la construction: $M(k, n) = (q, q')$.

Les consommations forment un monoïde avec $0 = (0, 0)$ et $(q, q') \cdot (p, p') =$

- $(q + p - q', p')$ si $q' \leq p$
- $(q, p' + q' - p)$ si $q' > p$

La composition de deux consommation est alors la consommation de la séquence. Dans le cas d'une analyse de WCET, les consommation sont $(q, 0)$ car on ne peut pas libérer de temps. On peut alors dénoter la consommation des structures de données, par exemple, $(e1, e2)$ a une consommation $M(\text{tuple}, 2) \cdot (q1, q1') \cdot (q2, q2')$

Cette sémantique est une version à grands pas de la ZAM, qui est plus proche de notre système de ressource, et donc facilite les preuves. On ne tient pas compte des différences entre ZAM et ZAM2, ou entre *push-enter* et *eval-apply*. Mais des métriques appropriées peuvent palier à cela.

Système de type affine avec pile

On introduit les *types simples*: unité, types numériques, références, tuples, variables de types (le tout classique), mais aussi *fonctions simples* et *types inductifs simples*:

Types des fonctions simple

Plutôt que d'avoir un type $t \to t$ dans la syntaxe, on a $[t_1 ... t_n] \to t$ qui spécifie que les n arguments sont appliqués en même temps. On a donc $\lambda f, x, y.fxy : ([T, U] \to V) \to T \to U \to V$ mais $\lambda f, x, y.(fx)y : (T \to U \to V) \to T \to U \to V$. On remplacera parfois $[T_1 ... T_n]$ par une unique pile Σ pour les jugements de typages.

Types inductifs simples

Dans un système normal, on défini les types algébriques comme: $T = \eta X.\langle C_1 : T_1 ... C_n : T_n \rangle$ avec X liée dans les T_i. Pour prendre en compte la taille des composantes de type X, il faut en connaître le nombre. Dans notre système affine, les types inductifs sont: $T = \eta X.\langle C_1 : (T_1, k_1) ... C_n : (T_n, K_n) \rangle$, avec X non liée dans les T_i. On emballe avec un k_i valeurs de types X, de sorte à ce que $(T, k) = T \times X \times ... \times X$ soit équivalent (dans un sens homotopique) à sa version classique.

Partage et polymorphisme

Le *let-polymorphisme* consiste à remplacer les valeurs quantifiés par des types par leur définition durant le typage pour pouvoir instancier sans conflit. Mais ici ce n'est pas une bonne idée: on change la consommation de ressource! RAML ne supporte que let-polymorphisme des fermetures. Il suffit de les re-définir autant que nécéssaire, de marquer les copies comme gratuites, et d'intancier les variables de types (maintenant des variables différentes) sans conflit.

Jugements et dérivations

Les jugements sont de la forme $\Sigma, \Gamma \vdash e : T$, avec Σ une pile de type rendant compte de l'état de la pile de la ZAM durant l'évaluation. Ainsi, il existe des jugements *fermés* pour lesquels $\Sigma = \emptyset$, qui représentent les jugement des valeurs elle-mêmes fermés, et de jugements *ouvert*, présent comme noeuds internes des dérivations, où $\Sigma \neq \emptyset$. On a deux théorèmes importants:

> Si e est bien typée, il n'existe qu'une seule dérivation de $\emptyset, \Gamma \vdash e : T$
>
> Pour tout Γ, Σ, T, si e est bien typée, il y au plus une seule dérivation de $\Sigma, \Gamma \vdash e : T$. Alors, on a aussi $\emptyset, \Gamma \vdash e : \Sigma \to T$, et la première dérivation ne peut apparaître qu'en sous-dérivation de la première.

On garantie ces unicités en produisant une règle de typage par règle d'évaluation.

On définit $H \vDash l \mapsto a : A$ signifiant "Dans le tas H, la position l contient une valeur de type A représentée en mémoire par a". En forme simple, $H \vDash l : A$, et en étendant point-par-point, $H \vDash S : \Sigma$ et $H \vDash V : \Gamma$ pour les piles et les ensembles de valeurs. On a alors,

> Si $H \vDash S : \Sigma, V : \Gamma$
> et $\Sigma; \Gamma \vdash e : T$
> et $S, V, H \vdash_M e \Downarrow (l, H') | (q, q')$
> alors $H' \vDash S : \Sigma, V : \Gamma, l : T$

Ce qui veut dire qu'en gros, tout va bien jusqu'à maintenant.

Polynômes de ressources

Polynômes de base

On veut, pour un type simple, définir les polynômes de base que l'on va combiner. Pour se faire, on interprète les types simple T en tant qu'arbres B_T. Les, types numériques sont des nombres, les types inductifs des noeuds internes et le reste des feuilles. Encore une fois, on ignore le coût des fermetures, que l'on a simplifié.

On va définir des abstractions en arbres des représentations mémoires dans le tas. Les arbres $b \in B_T$ d'un type inductif simple T dont les constructeurs sont $(C_i : (T_i, n_i))_{i \leq k}$ sont définis inductivement: les feuilles sont les types non-inductifs, et les noeuds internes sont $C_i(a_i, b_1, ..., b_{n_i})$ avec $a_i \in B_{T_i}$ et $b_j \in B_T$.

Pour un arbre $b \in B_T$, on peut considérer une descente dans cet arbre, où l'on ignore ou collecte des étiquettes. Il en résulte une suite finie $A = (a_m : T_m)_{m \leq k}$. On défini $\tau_B(b)$ l'ensemble de ces suites. Muni d'une suite $D = (D_m)_{m \leq k}$ de constructeurs de B, on note $\tau_B(D, b)$ l'ensemble des suites où a_i est l'étiquette d'un D_i-noeud. Armé de $\tau_B(D, b)$, on peut *enfin* définir les polynômes de bases.

Les *polynômes de bases* $P(T)$ d'un type simple T sont un ensemble de fonction des arbres de B_T vers les rationnels définis inductivement par:

- Pour tout T, $\lambda a.1 \in P(T)$
- Pour $T = T_1 \times ... \times T_n$, si $p_i \in P(T_i)$ alors $\lambda a_1 ... a_n . \prod p_i(a_i) \in P(T)$
- Pour $A = (a_m : T_m)_{m \leq k}$ résultante d'une descente, et
- Pour $T = \langle (C_i : (T_i, n_i)) \rangle$, pour toute suite D de constructeurs de B_T, on a $\lambda b. \sum_{A \in \tau_B(D,b)} \prod_{m \leq k} p_m(a_m)$ dans $P(T)$ dès que $\forall m, p_m \in P(T_m)$.

Les polynômes de base de T sont indicés par la famille $I(T)$ définie inductivement par:

- Pour tout T, $\quad \in I(T)$
- Pour $T = T_1 \times ... \times T_n$, si $i_j \in I(T_j)$ alors $(i_1, ..., i_n) \in I(T)$
- Pour T type inductif de constructeurs $(C_j : (T_j, n_j))_{j \leq k}$, toute séquence $[(C_{j_m}, i_m)]_{m \leq k}$ est dans $I(T)$ dès que $\forall m, i_m \in I(T_{j_m})$.
- On identifie , (, ...,), et [].

On associe naturellement un polynôme de $p_i \in P(T)$ à tout indice de $i \in I(T)$.

Polynômes de ressources d'un type

Pour un type T, les polynômes de ressources $\sum q_i \bullet p_i$ sont les combinaisons linéaires (de support fini) à facteurs rationnels de polynômes de base indicés par $I(T)$. Ils sont ainsi représentes par les séquences $(q_i)_{i \in I}$. On notera qu'ils sont croissants en chacun de leurs composantes, et on note $R(T)$ l'ensemble des polynômes, et $Q(T)$ l'ensemble de leurs réprésentations.

Le degré d'un polynôme de ressource est au plus celui d'un de ses polynôme de base, qui lui-même est calculable depuis son indice. On peut alors limiter

l'espace de recherche de l'analyse en limitant les indices. *RAML* par exemple impose les indices tels que les polynômes de ressources inférés soient de degrés au plus K, une constante définie par l'utilisateur.

Enfin, il sera bon de noter que deux indices peuvent dénoter le même polynôme de base, ce qui ne blesse pas la correction de l'analyse, et que certains indices peuvent représenter des polynômes nuls. Ces derniers peuvent être détectés et éliminés par leurs indices simplement.

Potentiel d'un programme

On va annoter les types simple avec des polynômes de ressources. A partir de maintenant, T réfère au type et $|T|$ à sa version simple, sans annotation, et ce inductivement. On peut spécifier les consommation des fonctions $f : T_f = [A_1, ..., A_n] \to B$ par des annotations Θ qui sont des ensemble de paires (Q_A, Q_B) avec $Q_A \in Q(|A_1 \times ... \times A_n|)$ et $Q_B \in Q(|B|)$. Dans les types, on a donc $\mathrm{f} : \langle \mathrm{T_f}, \Theta \rangle$. On nomme *type annoté* une paire (A, Q) d'un type A et d'un polynôme de ressource de $Q(|A|)$.

On peut alors définir le *potentiel* d'une valeur v d'un tas H. Si pour v, $H \vDash l \to a : |A|$, et que $v : (A, Q)$, son potentiel est

$$\Phi_H(v : (A, Q)) = \sum q_i \bullet p_i(a)$$

On étends point-à-point ce potentiel aux contextes et pile, en les entendant comme un produit cartésien des indices de leurs éléments. On note $I(\Sigma; \Gamma)$ les indices d'un contexte, et un *contexte annoté* est simplement $\Sigma; \Gamma; Q$. On définit le potentiel de l'état d'un programme comme tel:

Soit H un tas, S une pile et V en environnement tels que $H \vDash V : \Gamma, S : \Sigma$. On déconstruit $\Gamma = x_1 : A_1, ..., x_n : A_n$ avec $H \vDash V(x_j) \mapsto a_j : |A_j|$, et $S = l_1, ..., l_m$ avec $H \vDash l_k \mapsto b_k : |B_k|$. Le potentiel est la combinaison linéaire sur $I(\Sigma, \Gamma)$

$$\Phi_{S,V,H}(\Sigma; \Gamma; Q) = \sum q_i \bullet \prod p_j(a_j) \prod p_k(b_k)$$

Typage de ressources

Les règles de typages sont un médium peu adapté à la communication des calculs d'annotations que l'on présente dans cette partie. On choisi de les omettre. On présente néanmoins les opérations définis pour établir les jugements de type annotés.

(Dé)-construction des termes

On considère ici la transformation des potentiels issus de la construction/destruction d'un $x : B = \eta X.\langle ..., C : (A, n), ... \rangle$. Si le constructeur de x est

C, et que $y : A \times B^n$ (avec B^n les n-uplets de B), alors x et y sont équivalents et l'on passe de l'un à l'autre lors de l'application de C ou d'un filtrage par motif.Pour préciser les changements d'annotations que cet isomorphisme induit, on définit le C-dépliage d'annotations de potentiels.

On pose Σ et $\Gamma, x : B$ le contexte exécution *plié* et Q son annotation, et $\Gamma' = \Gamma, y : A \times B^n$ l'environnement *déplié*. L'annotation associée à $\Sigma; \Gamma'$ est notée $\lhd_B^C(Q)$, et est définie afin de conserver le potentiel total entre les deux contextes d'évaluation. On peut heureusement définir ce dépliage par contraintes linéaires.

Partage

Comme nous utilisons le partage, il nous faut aussi pouvoir combiner les annotations que $x_1 : A$ et $x_2 : A$ quand elles sont le résultat de `let share x as (x_{1}, x_{2}) in`. On prendra soin de remarquer que les usages de `x_1` et `x_2` ne sont en général pas symétriques, et qu'il faut donc coefficienter correctement les participations de `x_1` et `x_2` au potentiel de `x` dans l'annotation de la forme `let share` et de ses sur-formes. Encore une fois, ces coefficients sont décrits par contraintes linéaires. Si Q est l'annotation sous le `let share`, on note $\curlyvee Q$ au dessus.

Si vous regardez ce document eu format HTML, l'opérateur ne rendra pas correctement. Il s'agit d'une flèche vers le bas, avec deux branches en guise de corps, l'une courbant vers la gauche, et l'autre vers la droite

Sous-typage

On définit le sous-typage structurel classique sur les types annotés, avec pour les fonctions, la condition qu'un type $A \to B, \Theta$ est sous-type de $A' \to B', \Theta'$ si A' est un sous-type de A, B en est un de B' et $\Theta' \subset \Theta$.

Jugement

Le jugement du système de ressourece est $\Sigma; \Gamma; Q \vdash_M e : (A, Q')$ s'entendant comme "Selon la métrique de ressourec M, dans un contexte d'évaluation $\Sigma; \Gamma$ (distingués pour leurs coûts) annotés par Q, si on a disposition au moins $\Phi(\Sigma; \Gamma; Q)$ de potentiel, alors l'expression e a le type $|A|$, et soit elle diverge, soit elle produit une valeur v, et dans ce cas le potentiel restant est d'au moins $\Phi(v : (A, Q'))$".

On introduit la métrique cf de coût nul pour les constantes, purement utilitaire, qui intervient dans la règle *LET*.

Règles

C'est un gros morceau, qui est certainement plus clair en algorithme qu'en code. On le passe pour le moment.

Théorème: Cohérence

Si $H \vDash V : \Gamma, S : \Sigma$ et $\Sigma, \Gamma, Q \vdash_M e : (B, Q')$:

1. alors $S, V, H \vdash_M e \Downarrow w|(p, p')$ et $p \leq \Phi_{S,V,H}(\Sigma, \Gamma, Q)$,
2. et si, de plus $w = (l, H')$:
   (a) $H \vDash l : B$,
   (b) et $p - p' \leq \Phi_{S,V,H}(\Sigma, \Gamma, Q) - \Phi_{H'}(l : (B, Q'))$

Preuve par double induction: sur le jugement de réduction, et au delà dans le jugement de typage.