Notes de lecture: Compositional Certified Resource Bounds

Cet article introduit l'analyse/C^{4}B/ pour la dérivation de bornes aux-pire-cas de consommation de ressource des programmes *C*. Par rapport à l'état de l'art de l'époque (2015), elle est *composable*, *interactive*, *certifiante*, et *efficace à l'échelle*. Les travaux présentés ici serviront de base pour la thèse de Carbonneaux.

Introduction

Motivation

Voir l'abstract de file:carbonneaux2018.org.

Au delà des avantages familiers de l'analyse de ressource les programmes exécutés sur des plate-formes non-fiables doivent pouvoir bénéficier de garanties statistiques sur leur réussite. Si l'opération matérielle P échoue avec probabilité p, une boucle contenant P a une change d'échec d'au moins p^k, avec k une borne du nombre de tours. Obtenir des informations statiques fines sur l'exécution du programme affine donc les informations sur la confiance à accorder aux systèmes non-fiables.

Enfin, l'interaction avec l'utilisateur est un espace des possibles: Les analyses de ressources considérées sont consistantes, donc incomplètes. Il est alors pertinent de permettre à un être humain de donner les dernières informations permettant à l'analyse de conclure. Notons bien: si l'interaction veut être compatible avec la génération de certificats, il faut automatiquement garantir que les "indices" fournis par l'humain soient en suite re-dérivables par le système d'analyse: l'humain donne des coupures, pas des axiomes.

Analyses "impératives"

Les outils d'analyse statique de ressources forment deux grandes familles, que l'on nommera *impératives* et *fonctionnelles*, par abus. Les analyses *impératives* permettent d'obtenir des bornes très fines sur des programmes impératifs structurés entiers, mais elles composent relativement mal. Il est délicat d'abstraire une procédure à ces appelants, ou encore d'analyser une séquence de boucles

sans avoir à calculer les pre-/post- conditions arithmétiques entre elles. Et c'est *beaucoup* plus facile à dire qu'à faire.

Les travaux de cette branche de l'analyse statique automatique de ressource ont produits des outils comme *SPEED*, *PUBS*, *LOOPUS*, *Rank* et *KoAT* (voir file:carbonneaux2018.org). Elles sont basées sur les *fonctions de classement* et les *registres virtuels*, qui sont une source majeurs de non-composition. L'inlining est donc de mise, et avec lui la redondance, et donc l'absence de passage à l'échelle.

Analyses "fonctionnelles"

La seconde forme d'analyse est basée sur l'analyse amortie (méthode du potentiel) et les systèmes de types. Les invariants de classes en OOP, et ceux des structures de données dans la logique de séparation peuvent être vues comme des types dans des systèmes pertinent pour l'analyse de ressource.

Les analyses *fonctionnelles* sont elles hautement composables, et peuvent fonctionner directement sur la syntaxe: on parle d'analyse dirigée par les types. Mais elles reposent sur l'obtention de décroissance sur un ordre bien-fondé, qui peut souvent leurs échapper. Par exemple, dans l'itération `for(i:=n; i++; i<m)`, la décroissance est sur l'expression linéaire `m-i`. Les designs patterns impératifs courants comme le flux de contrôle non-linéaire ou la mutation ont formé jusqu'à peut des obstacles insurmontables aux analyses *fonctionnelles*.

Contribution

C^{4}B est la première analyse *semi-automatique* de ressource. Nous avons adapté les méthodes à potentiels pour inférer l'usage de ressource de programmes C comprenant, entre autres, les structures suivantes:

- les fonctions mutuellement récursives;
- les boucles en séquence;
- les boucles imbriqués partageant des indices;
- la libération mémoire dynamique.

Techniquement, l'analyse porte sur *Clight*, la première représentation intermédiaire de *CompCert*. C'est un sous-C avec une seule structure de boucle, et sans effets de bords dans les expressions.

Notre système est basé sur une logique à la Hoare portant sur des combinaisons de variables. Il est le premier à gérer les bornes dépendants de variables `signed` et de différences entre variables. Il permet la composition d'analyse, notamment en générant des abstractions de ressources pour les fonctions, et produit des certificats pour les bornes inférées vérifiables en temps linéaire.

Les bornes sont produites en réduisant les relations linéaire et amorties non-linéaires entre les variables à un problème de programmation linéaire classique,

une nouveauté. Notre analyse a été prouvée cohérence par rapport à une sémantique paramétrique de coût de C, qui peut être modifiée à posteriori (fonction `tick(n)`).

Nos avons testé C^{4}B sur plus de 2900 lignes de code C et l'avons comparé dans un cadre expérimental avec d'autre analyseurs de ressource existants. Le benchmark *cBench* à servi de banc de test, augmenté de plusieurs extraits de projets open source et libres ayant démontrés leurs résistance à l'analyse statique. Il en ressort, avec les précautions d'usage, que C^{4}B est le seul capable de d'inférer à la fois des bornes globales sur des programmes de grandes tailles et des bornes locales sur des flots de contrôles complexes.

Méthode du potentiel

Analyse quantitative

On commence avec une *sémantique opérationnelle* à grands pas d'un langage impératif idéal: un programme S munit d'un ensemble d'état initial $\mathbb{S}$ et d'un état initial σ, s'exécute avec un coût n et produit un état final σ' et on note cela $(S, \sigma) \Downarrow_n \sigma'$. La méthode du potentiel consiste à définir une fonction Φ des états vers $\mathbb{Q}^+$ et de prouver la

> *Propriété des potentiels*:
> Si $\Phi(\sigma) \geq n$, alors $(S, \sigma) n \sigma'$ pour un certain σ'.

$\Phi(\sigma)$ est alors une borne statique acceptable de la consommation de ressource. On peut ensuite définir des triplés à la Hoare pour les fonctions de potentiels:

> $\{\Phi\} S \{\Phi'\}$ si et seulement si, pour tout σ, σ', n:
> $(S, \sigma) \Downarrow n\sigma'$ implique $\Phi(\sigma) \geq n + \Phi'(\sigma')$

On retrouve alors une règle classique de la logique de Hoare, mais Φ' n'est définie qu'avec l'aide de S, un avantage clair.

$$\frac{\{\Phi\} S \{\Phi'\} \qquad \{\Phi'\} S' \{\Phi''\}}{\{\Phi\} S; S' \{\Phi''\}}$$

Enfin, pour s'éviter une lourdeur syntaxique, nous notons $\langle x, y \rangle$ la valeur $max(y - x, 0)$ (que le papier note $|[x, y]|$, la norme positive de l'intervalle $[x, y]$).

Abstraction du programme

Les analyses quantitatives ne suffisent pas à raisonner sur les programmes. Il faut maintenir *à minima* un état abstrait justifiant les opérations sur les fonctions de potentiels. Notre outil représente cette état comme un contexte logique, extrait d'une interprétation abstraite. Le domaine de choix est celui des polyèdres, avec accélération importante des points fixes: des invariants locaux faibles sont empiriquement suffisant.

Exemples juteux

Je conseille d'essayer de trouver, à la main, le coût de ces fragments de code avant de lire la réponse exacte, inférée correctement par C^{4}B. On passe les exemples sur les fonctions mutuellement récursives et les itérations tordues.

Quelques itérations classiques:

```
/* (1) */ while(x < y) {x++; tick(1)}
/* (2) */ while(x+K <= y) {x += K; tick(T)}     // K et T sont des constantes
/* (3) */ while(x < y) {tick(-1); x++; tick(1)}
```

Parmi les outils testés, tous bornent correctement le coût $\langle x, y\rangle$ pour *(1)*, mais seul C^{4}B infère la borne exacte $\frac{T}{K}\langle x, y\rangle$ pour *(2)*, qu'il traite de manière totalement identique. Ce n'est pas le cas pour les autres. L'exemple *(3)* est de coût nul, ce qu'une méthode basé sur les méthodes de classements ne peux établir. Nos fonctions de potentiels elles infèrent bien que le coût du programme est 0.

Itérations imbriquées interdépendantes

```
while (n<0) {
    n++;
    y = y+1000;
    while(y>=100 && rand_bool()) {
        y = y-100;
        tick(5);
    }
    tick(9)
}
```

La borne correcte est $59\langle n, 0\rangle + 0.05\langle 0, y\rangle$. En effet, la valeur initiale de `y` n'influe que sur la première évaluation de la boucle.

Traitement par bloc

```
for( ; l>=8; l-=8)
    tick(N);  // N est une constante
for( ; l>0; l--)
    tick(1)
```

C'est un schéma classique de traitement par bloc, qui a été retrouvé par les auteurs dans tout les chiffres par blocs implémentés par *PGP*, dans les manipulation de bits de *libtiff*, ou encore dans un encodeur *MPEG*. Le raisonnement est le même pour la machine et l'humain avec d'autres constantes que 8 et 1. Il faut en fait sublimement équilibrer les potentiels entre les deux boucles en fonction de la taille de `l` par rapport à `N`. Le coût exacte est ici $\frac{N}{8}\langle 0, l\rangle$ quand $N \geq 8$, et $\frac{7}{8}(8 - N) + \frac{N}{8}\langle 0, l\rangle$ quand $N < 8$. Notre outil est le seul pouvant rendre compte des bornes précises pour toutes les valeurs de N.

Potentiels linéaires

Les fonctions de potentiel utilisées sont les combinaisons linéaires, à coefficients rationnels. Avant de proprement définir Φ nous devons définir les indices et éléments de bases des combinaisons linéaires. On travaille à programme fixe, et on définit:

- L l'ensemble des variables locales à un point du programme;
- G l'ensemble les variables globales;
- C un ensemble fini de *symbole de constante*.

On suppose ces trois ensembles sont disjoints.

On définit *l'état simplifié du programme* $\sigma : (G \cup L \cup C) \to \mathbb{Z}$ associant aux variables et aux constantes leurs valeurs abstraites entières. L'ensemble des indices des fonctions de potentiels est:

$$I = \{(x, y) | x, y \in (G \cup L \cup C) \wedge x \neq y\}$$

L'ensemble des fonctions de potentiel de base associe donc à une paire d'entité du programme le potentiel qu'elle engendre. Ce potentiel dépend évidemment de l'indice $i \in I$ considéré et de l'état courant σ. On définit, pour $i = (x, y)$:

$f_i(\sigma) = \langle \sigma(x), \sigma(y) \rangle = max(\sigma(y) - \sigma(x), 0)$ si $i \neq 0$
$f_0(\sigma) = 1$

On peut enfin définir les fonctions de potentiels de notre analyse.

$\Phi : \mathbb{S} \to \mathbb{Z}$
$\Phi(\sigma) = q_0 + q_{(x,y)} \langle \sigma(x), \sigma(y) \rangle + q_{(y,z)} \langle \sigma(y), \sigma(z) \rangle + ...$ où les indices parcourent I

On utilisera les formes courtes de ses fonctions: on inclut un symbole I_0 de 0 dans I, que l'on rend implicite: Les fonctions de potentiel sont entièrement définies par la famille $Q = (q_i)_{i \in I}$ des coefficients. Comme I est fini (chaque programme l'est), Q est donc une *famille de rationnels positifs de support fini*, qui est une construction finitiste !

$\Phi(\sigma) = \sum_i q_i f_i(\sigma) = Q(\sigma)$

Système de type

La sémantique de *Clight* est doté d'une *métrique de ressource* M qui donne le coût dans $\mathbb{Q}$ de chaque étape de la sémantique opérationnelle. Dans nos règles, less sont limitées à `x ← y` et `x ← x ± y`. Pour ce faire, nous ré-écrivons le programme en explosant les affectations non-linéaires, en prenant soin d'adapter les q_i idoines pour conserver un coût identique. Ce pré-traitement est sûr et garantit *qu'aucune borne de boucle de dépend résultats d'opérations non-linéaires.*

Jugements de typage

On définit les jugements de typages quantitatifs pour ce langage: ils sont basés sur ceux de la logique de Hoare: toutes les propositions sont séparées en une partie logique Γ et une partie quantitative Q. Le jugement principal est:

$$(\Gamma_B; Q_B), (\Gamma_R; Q_R) \vdash \{\Gamma; Q\} S \{\Gamma'; Q'\}$$

> Pour un état initial σ, si $Q(\sigma)$ de potentiel est disponible et Γ vaut, on peut exécuter S sans débordement en produisant un état σ'; il reste au moins $Q'(\sigma')$ de potentiel et Γ' vaut.

Le jugement est en vérité un iota plus complexe, pour gérer les flux de contrôles non-linéaires: comme en logique de Hoare, $(\Gamma_B; Q_B)$ est la post-condition qui tient si S sort d'un boucle avec `break`, et $(\Gamma_R; Q_R)$ celle qui tient s'il "`return`".

Notre système de type collecte des informations sur les fonctions globales du programme dans un contexte implicite Δ. Pour une fonction `f` d'arguments formels et de valeur retour pour l'appelant, on a:

$$\Delta(f) = \lambda a.(\Gamma_f; Q_f), \lambda \rho.(\Gamma'_f; Q'_f)$$

> En instanciant les paramètres des informations de fonctions, on sait que dans un état σ, si $Q_f(\sigma)$ de potentiel est disponible et que Γ_f vaut, alors l'appel à f s'exécute sans débordement en produisant un état σ'; il reste au moins $Q'_f(\sigma')$ de potentiel et Γ'_f vaut.

Système de type

On utilise un système de type pour la logique de Hoare augmenté pour gérer le potentiel. Il faut donc *accumuler* le potentiel pour certaines instructions et le *dépenser*, après avoir vérifier que c'est possible, pour d'autre. L'analyse statique préalable permet de spécialiser les règles de typages. Par exemple, *INCP* gère les instructions `x ← x+y` quand on peut déduire que `y  0`, *INCN* quand `y  0`, et *INC* quand l'analyse de signe de `y` n'est pas concluante. Des symétries apparaissent tout le long de notre système de type.

Assignement

On considère l'instruction uniquement `x←x+y` avec `y 0`: les autres assignements linéaires sont identiques à symétrie près, quitte à perdre en précision. La règle correspondante est *INCP* et ses cousines sont *DECP*, *INCN*, *DECN*, *INC*, et *DEC*.

L'évaluation de `x+y` a un coût propre M_e(`x+y`), et l'affectation un coût M_u (pour *update*). On paye ces coûts avec le potentiel, mais comment évolue-t-il quand `x` augmente de `y` ? Par linéarité, les seuls termes du potentiel qui changent sont ceux des intervalles impliquant `x`, qui sont $\langle x, u\rangle$ et $\langle u, x\rangle$ pour tout les autres symboles u.

On fixe maintenant u. On note `x` la valeur de `x` après l'affectation. Nécessairement, $x \leq x$, donc $\langle x, u\rangle$ diminue. On peut alors récupérer son potentiel. Si de plus $x' \leq u$, le potentiel q_{0y} de `y` augmente d'au moins q_{xu}. Dans le cas contraire, on a $u \leq x'$, donc $\langle x, u\rangle = 0$ après l'affectation, et $\langle u, x\rangle$ augmente durant. On paie cette augmentation avec q_{0y}. En itérant la construction pour tout les u, on obtient un règle dirigée par la syntaxe.

Itération (règle *LOOP* et *BREAK*)

L'itération dans notre langage cible est toujours de la forme `loop { ... break ... }`. L'analyse doit trouver le potentiel Q qui peut payer pour la boucle. Le potentiel post-boucle est passé dans les préconditions "break" du jugement du corps pour éventuellement être utilisée pour la sortie, et on paie le coût M_l de la boucle à chaque itération. On est encore dirigé par la syntaxe: la post-condition du `break` est celle de la boucle qu'il termine.

Appels de fonctions: *CALL*

La règle est impressionnante, mais se résume à:

1. Instancier la spécification de `f` avec les arguments et le retour effectif;
2. Conserver les potentiels des intervalles de variables locales;
3. Mettre à zéro les potentiels dépendants de variables globales.

Affaiblissement et relaxation

L'affaiblissement classique est autorisé. On peut aussi volontairement demander plus de potentiel dans les pré-conditions et en libérer moins dans les post-conditions. Cela implique une notion de *plus/moins* de potentiel, donnée par la règle *RELAX*. La relaxation définit le pré-ordre sur les potentiels, mais, plus important, elle permet la communication entre l'analyse qualitative et quantitative: si on peut borner la valeur d'un intervalle, on peut rendre constant le potentiel correspondant sans perte de sûreté.

Inférence automatisée par programmation linéaire

L'analyse de ressource comporte deux étapes. La dérivation des contraintes quantitatives de potentiel est la première. On l'applique co-inductivement sur tout les composant fortement connexe du code,en utilisant les spécifications de fonctions obtenues pour les composantes suivantes. Les coefficients q_i sont tous symboliques, et les contraintes sont accumulés.

Dans un second temps, on utilise un solveur de programmation linéaire tiers pour obtenir les coefficients effectifs. On résout d'abord pour les coefficients indicés par des variables, puis ceux indicés par des constantes dans un second temps:

Pour les premiers coefficients, on demande de résoudre les contraintes collectées, mais on souhaite aussi minimiser les potentiels liés au variables libres, et cela plus que les ceux associés aux variables liés et au constantes. On fournit donc au solveur un objectif Obj de minimisation encodant nos choix de priorités. Comme on sait $\langle 10, x\rangle$ est plus précis que $\langle 0, x\rangle$ l'est plus que $\langle 0, 10\rangle$... on pose $Obj = 1 \cdot \langle 0, x\rangle + 1000 \cdot \langle 10, x\rangle + ...$ On obtient du solveur une solution pour les contraintes et une valeur minimale de l'objectif min.

On rajoute la contrainte $Obj \leq min$ et on demande à minimiser $\sum(y - x + 1) \cdot \langle x, y\rangle$. qui donne les bornes pour les coefficients des constantes. Tout ces coefficients feront parti du certificat que nous générons. On notera que si nous avions utilisé un solveur SMT, ce certificat aurait pris la forme, plus complexe, d'un contre exemple dans une théorie.

Contexte logique et interaction utilisateur

Le problème de l'arrêt pouvant se réduire à la recherche d'une fonction de potentiel, notre analyse est nécessairement incomplète. Il est alors judicieux de permettre une extension à posteriori de la partie logique de l'analyse, comme les métriques interchangeables le permettent pour la partie quantitative.

Dans cet article, l'approche retenue est l'ajout de *variables auxiliaires* et d'assertions aux programmes pour lesquels l'analyse automatique n'est pas suffisante. En liant les variables auxiliaires et les variables utiles dans les assertions, on peut étendre localement le contexte logique et permettre à l'analyse de progresser. Mais il faut pouvoir prouver (avec certificat) que l'ajout variables auxiliaires ne change pas la sémantique, et étendre l'outil pour permettre à l'utilisateur de décrire les invariants des variables auxiliaires. La figure 7 exige, par exemple, de décrire que la variable auxiliaire `na` est le nombre d'entrée dans un tableau de bit valant 1.

Les artefacts de l'article étant introuvables, la dernière version publique de C^4B précédant cet article de 5 ans, le rapport technique ne donnant pas de détail, il est impossible de décrire un fonctionnement précis. On pourra noter que, dans la thèse de Carbonneaux, celui-ci indique la possibilité d'annoter des programmes dans *Coq* avec des propositions du calcul des constructions inductives. On pourrait alors spéculer que des programmes *Clight* écrit en tant que littéraux Coq aurait pu servir d'artefacts. L'approche retenue dans la thèse est une approche modulaire de l'analyse. Le cas de la figure 7 est résolu par l'ajout d'un domaine logarithmique aux fonctions de potentiels.

Relation à l'état de l'art

l'auteur des notes invite une dernière fois de lire la partie ***Travaux Connexes*** *des notes sur la thèse de Carbonneaux.*

Nous avons été inspirés par l'analyse de potentiel pour les programmes fonctionnels selon Hoffmann, et y avons ajouté aux potentiels des variables individuelles les potentiels aux intervalles entre variables, mais aussi entre *variables auxiliaires*, qui sont clés dans l'analyse à la Hoare. Nous avons ainsi résolu le problème ouvert de l'analyse de ressource amortie automatique des boucles sans décroissance bien-fondée triviale.

L'interaction utilisateur est une nouveauté dans l'analyse amortie automatique, tout comme la possibilité de prendre en compte la libération dynamique de mémoire. Mais tout n'est pas rose: notre analyse ne gère pas les pointeurs de fonctions ou les pointeurs vers la pile pour des raison d'implémentation. Les points faibles de cette analyse sont, selon notre jugement:

- les invariants du tas (p.ex. les chaînes de caractères terminées par `\0`);
- l'usage de ressource dépendant non-trivialement d'opérateurs non-linéaires (`*`, `%`, …)
- les programmes dont la terminaison n'est prouvable que par des raisonnements complexes de flux de contrôle.

Dans la littérature:

- Hoffmann a déjà produit des analyses dont les fonctions de potentiel sont de la forme $\sum_{i,j \in I} m_i m_j$, permettant l'analyse fine l'algorithme de matrice encodées comme listes de listes.

  Note du lecteur: il est depuis allé beaucoup plus loin, voir TARBAC.

- Braberman & al produit des bornes polynomiales pour la consommation de tas d'appels de méthodes dans les langages à la Java par des méthodes de région, en passant par un problème d'optimisation polynomiale paramétrique, résolue par un solveur externe.