Lot 4.3

Technologie de modélisation

Complexité en espace et en temps

Un système d'analyse pour les programmes fonctionnels du premier ordre

Description :	La bonne exécution d'un programme nécessite un certain nombre de ressources qui sont mesurées en temps, en espace mémoire, ou encore en nombre d'appels récursifs. L'objectif est de déterminer ces ressources dans le cas d'un programme fonctionnel du premier ordre, comme ceux d'ELAN.
Auteur(s) :	Jean-Yves Marion
Référence :	Averroes / Lot 4.3 / Fourniture 2 / V1.0
Date :	26 Avril 2004
Statut :	à valider
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

1 avril 2003	V 0.1	version préliminaire
26 Avril 2004	V 1.0	mise au format averroes

Table des matières

1 Objectifs généraux, contexte et enjeux **3**

2 Méthodologie et autres approches **3**

3 Résultats : Analyse par quasi-interprétations **3**
3.1 Programmes fonctionnels 3
3.2 Quasi-interprétations 4
3.3 Calcul des ressources 4
3.3.1 Interprétation des constructeurs 4
3.3.2 Terminaison 4
3.3.3 Temps 4
3.3.4 Espace 5
3.3.5 Transformation de programmes 5
3.4 Prototype ICAR 5

4 Directions **6**

1 Objectifs généraux, contexte et enjeux

Le développement de systèmes soulève de nouvelles questions de sécurité liées aux propriétés quantitatives sur les ressources engagées lors de l'exécution d'une application. Prenons des exemples:

- Un serveur ne prend en compte une requête d'une application que si le temps de calcul est acceptable. Pour cela, le serveur demandera à l'application cliente un "certificat de complexité".
- Un utilisateur veut avoir la certitude que l'application qu'il charge sur son assistant personnel ou son téléphone mobile pourra s'exécuter avec la mémoire disponible, et sans bloquer, par manque de ressources, les autres applications. A savoir, un programme qui demande trop de ressource est potentiellement un programme qui attaque un système.
- Un fournisseur de composants logiciels pour un système embarqué veut démontrer que son produit répond aux spécifications en terme de ressources mémoire ou de temps de réponse, qui sont imposées par le constructeur.

Ces exemples illustrent l'importance de la maîtrise des ressources utilisées par un système, bien que ces propriétés aient été sous-estimées jusqu'à présent.

L'objectif est de déterminer les ressources nécessaires à un système, en temps d'exécution, en espace mémoire, en nombre d'appels récursifs d'une fonction, ...

2 Méthodologie et autres approches

Nous avons cherché à développer des méthodes statiques d'analyse. Ces méthodes d'analyse, comme le typage, vérifie un programme avant son exécution. Il y a au moins deux raisons qui justifient cette approche par rapport aux autres. D'une part, cela nous permet de produire un certificat de garantie de ressources pour un programme qui peut être transmis au client d'une application comme dans le contexte de "Proof carrying Code". Et d'autre part, le programme n'est pas ralenti par un contrôle des ressources durant l'exécution. Une autre approche d'analyse statique consiste à établir une borne disons sur le temps de calcul en posant des équations qui comptent le nombre d'étapes de calcul, voir le travail [3]. Un défaut de cette méthode est l'impossibilité de transformer un programme pour accélérer son exécution comme nous l'illustrerons.

3 Résultats : Analyse par quasi-interprétations

Le reste du rapport résume les travaux recherche que nous avons menés. Ils sont détaillés dans le récent état de l'art [4].

3.1 Programmes fonctionnels

Un programme fonctionnel du premier ordre est représenté par un système de réécriture comme dans l'exemple du tri par insertion ci-dessous.

$$
\begin{array}{ll}
\texttt{if}\ \ \mathbf{tt}\ \texttt{then}\ \ x\ \texttt{else}\ \ y & \to x \\
\texttt{if}\ \ \mathbf{ff}\ \texttt{then}\ \ x\ \texttt{else}\ \ y & \to y \\
\mathbf{0} < \mathbf{S}(y) & \to \mathbf{tt} \\
x < \mathbf{0} & \to \mathbf{ff} \\
\mathbf{S}(x) < \mathbf{S}(y) & \to x < y \\
\texttt{insert}(a,\mathbf{nil}) & \to \mathbf{cons}(a,\mathbf{nil}) \\
\texttt{insert}(a,\mathbf{cons}(b,l)) & \to \texttt{if}\ \ a < b\ \texttt{then}\ \ \mathbf{cons}(a,\mathbf{cons}(b,l)) \\
 & \qquad \texttt{else}\ \ \mathbf{cons}(b,\texttt{insert}(a,l)) \\
\texttt{sort}(\mathbf{nil}) & \to \mathbf{nil} \\
\texttt{sort}(\mathbf{cons}(a,l)) & \to \texttt{insert}(a,\texttt{sort}(l))
\end{array}
$$

Un programme fonctionnel calcule une fonction définie sur un domaine engendré par un ensemble de constructeur. Dans l'exemple, les constructeurs sont **tt**,**ff**,**cons**,**nil**,**S**, et **0**. Ainsi le domaine de calcul est l'ensemble des booléens et les listes sur les entiers.

3.2 Quasi-interprétations

Les quasi-interprétations ont été proposées par Marion [7], dans la thèse de Bonfante, et dans Marion et Moyen [8].

Une quasi-interprétation polynomiale (abrégée QI) d'un symbole de fonction f d'arité n est une fonction $[\![f]\!] : \mathbf{R}^{+^n} \to \mathbf{R}^+$ qui vérifie

- $[\![f]\!]$ est bornée par un polynôme
- $[\![f]\!](X_1, \cdots, X_n) \geq X_i$ pour $1 \leq i \leq n$.
- $[\![f]\!]$ est croissante par rapport à chaque variable.

Les QIs sont étendues aux termes : $[\![f(t_1, \cdots, t_n)]\!] = [\![f]\!]([\![t_1]\!], \cdots, [\![t_n]\!])$.

$[\![-]\!]$ est une QI d'un programme `main` si pour chaque règle $l \to r$ et pour chaque substitution σ, $[\![l\sigma]\!] \geq [\![r\sigma]\!]$.

Les QI s'apparentent aux interprétations polynomiales. A la différence de celles-ci, la terminaison n'est pas assurée par une QI. De plus, une QI n'est pas forcément un polynôme comme l'illustre l'exemple `lcs` du paragraphe 3.3.5.

Le programme du tri par insertion `sort` a la QI $[\![-]\!]$ suivante

- $[\![\mathbf{tt}]\!] = [\![\mathbf{ff}]\!] = [\![\mathbf{0}]\!] = [\![\mathbf{nil}]\!] = 1$
- $[\![\mathbf{S}]\!](X) = X + 1$
- $[\![\mathbf{cons}]\!](X,Y) = [\![\mathtt{insert}]\!](X,Y) = X + Y + 1$
- $[\![\mathtt{sort}]\!](X) = X$

3.3 Calcul des ressources

La mesure de la complexité d'un programme est liée à deux critères que nous allons détailler.

3.3.1 Interprétation des constructeurs

Les QIs des programmes sont classées suivant le type de QIs données aux constructeurs.

- **Type 0** Si $[\![\mathbf{c}]\!](X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + b, b \geq 1$;
- **Type 1** Si $[\![\mathbf{c}]\!]$ est un polynôme dont le degré de chaque variable est au plus 1;
- **Type 2** Si $[\![\mathbf{c}]\!]$ est n'importe quel polynôme.

Maintenant, supposons que f admette une QI de type 0. Soient u_1,...,u_n des éléments du domaine de f, i.e. des termes constructeurs. Une propriété important est que si $f(u_1,...,u_n)$ a pour valeur u alors la taille de u est bornée par un polynôme dans la somme des tailles des entrées u_1,...,u_n.

3.3.2 Terminaison

L'autre outils de calibration est la preuve de terminaison associée au programme. Il faut observer qu'un programme qui possède une QI ne termine pas forcément. Nos travaux nous ont conduit à considérer deux ordres de terminaison, très connus, qui sont MPO (Multiset path ordering) et LPO (Lexicographic path ordering). Ainsi, le programme du tri par insertion `sort` termine par MPO.

3.3.3 Temps

Nous avons démontré que l'ensemble des fonctions calculables par un programme qui termine par MPO est

- l'ensemble des fonctions calculables en temps polynomial quand l'interprétation des constructeurs est de type 0;

- l'ensemble des fonctions calculables en temps exponentiel quand l'interprétation des constructeurs est de type 1;
- l'ensemble des fonctions calculables en temps doublement exponentiel quand l'interprétation des constructeurs est de type 2.

Les réciproques sont vraies, ce qui donne de nouvelles caractérisations des classes de complexité en temps.

De plus, nous avons aussi considéré des spécifications écrites par des règles de réécriture. Autrement dit à la différence du propos précédent où les programmes étaient des systèmes de réécriture confluents, nous considérons des systèmes non-confluents. Ceci nous a conduit à caractériser la fameuse classe NP des problèmes résolus en temps polynomial sur machines de Turing non-déterministes.

Les QIs de type 0 ont recueilli le plus gros de nos efforts car ils cernent au plus près ce qui est effectivement calculable.

3.3.4 Espace

Dans le cas de l'espace, nous obtenons un résultat fort utile qui est le suivant. Si un programme termine par LPO (ou MPO) alors l'espace de calcul est borné par un polynôme dans la taille des entrées du programme. Une conséquence est alors que la taille de la pile d'appels récursifs est aussi bornée par un polynôme dans la taille des entrées du programme. Un théorème de hiérarchie analogue à celui sur le temps mentionné plus haut a été établi.

3.3.5 Transformation de programmes

Le programme `lcs` calcule la longueur de la plus longue sous suite commune de deux mots sur $\{a,b\}^*$, comme on le trouve dans le très classique livre [6] sur l'algorithmique.

$$
\begin{array}{lll}
\texttt{lcs}(\boldsymbol{\epsilon},y) & \to \mathbf{0} & \\
\texttt{lcs}(x,\boldsymbol{\epsilon}) & \to \mathbf{0} & \\
\texttt{lcs}(\mathbf{i}(x),\mathbf{i}(y)) & \to \mathbf{S}(\texttt{lcs}(x,y)) & \mathbf{i} = \mathbf{a},\mathbf{b} \\
\texttt{lcs}(\mathbf{i}(x),\mathbf{j}(y)) & \to \texttt{max}(\texttt{lcs}(x,\mathbf{j}(y)),\texttt{lcs}(\mathbf{i}(x),y)) & \mathbf{i} \neq \mathbf{j} \\
\texttt{max}(n,\mathbf{0}) & \to n & \\
\texttt{max}(\mathbf{0},m) & \to m & \\
\texttt{max}(\mathbf{S}(n),\mathbf{S}(m)) & \to \mathbf{S}(\texttt{max}(n,m)) &
\end{array}
$$

Le programme[1] `lcs` termine par MPO avec la précédence $\texttt{max} < \texttt{lcs}$ et admet la QI suivante de type 0.

- $[\![\boldsymbol{\epsilon}]\!] = [\![\mathbf{0}]\!] = 1$
- $[\![\mathbf{a}]\!](X) = [\![\mathbf{b}]\!](X) = [\![\mathbf{S}]\!](X) = X + 1$
- $[\![\texttt{lcs}]\!](X,Y) = [\![\texttt{max}]\!](X,Y) = \max(X,Y)$

Cet exemple est particulièrement excitant. Il n'est pas trop difficile de voir que le nombre d'appels récursifs est exponentiel dans la taille des mots d'entrée. Autrement dit, l'évaluation par appel par valeur nécessite un temps exponentiel. Cependant le résultat mentionné affirme que le programme `lcs` définit une fonction calculable en temps polynomial. Une conséquence du résultat obtenu est une stratégie de transformation automatique de programme par mise en cache des valeurs. De plus, nous garantissons que seules les valeurs nécessaires sont mémorisées à une étape donnée du calcule. Cette technique s'inspire des travaux de N. Jones [2] et en particulier de sa relecture de la simulation en temps polynomial d'un automate à pile bilatère par Cook [5].

3.4 Prototype ICAR

Nous avons réalisé un logiciel ICAR en Ocaml. ICAR est un interpréteur d'un langage fonctionnel du premier ordre qui vérifie la terminaison par MPO et vérifie que le programme a une QI

1. Le programme `lcs` n'est pas déterministe mais il est bien confluent, ce qui ici suffit

donnée. La vérification est sous-traitée à MUPAD. Le programme peut passer d'une sémantique par appel par valeur à une sémantique par cache. Ce prototype, et bien d'autres choses, est décrit dans la thèse de JY Moyen [9].

4 Directions

Nous suivons actuellement trois directions pour compléter et étendre l'approche décrite dans ce rapport.

1. L'une des difficultés est de trouver une QI d'un programme, ce qui pose deux problèmes. Le premier est d'associer à chaque symbole de fonction une QI bornée par un polynôme. Le second est de vérifier les inégalités entre QI. Des travaux ont été fait dans cette direction [1].
2. Nous travaillons sur la d'un prototype qui comprend un compilateur de langage fonctionnel du premier ordre, une machine virtuelle qui exécute le bytecode produit par le compilateur. La machine virtuelle est en charge de la vérification de la taille de la pile par cette technique de QI.
3. Le dernier point vise à analyser directement un bytecode. Pour cela, nous avons proposé une interprétation d'un programme par l'intermédiaire d'un réseau de Petri. Nous avons ainsi caractérisé une classe de programme qui s'exécute sans allouer de mémoire supplémentaire.

Références

[1] R. Amadio. Max-plus quasi-interpretations. In M. Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2003.

[2] N. Andersen and N.D. Jones. Generalizing Cook's transformation to imperative stack programs. In J. Karhumäki, H. Maurer, and G. Rozenberg, editors, *Results and trends in theoretical computer science*, volume 812 of *Lecture Notes in Computer Science*, pages 1–18, 1994.

[3] R. Benzinger. *Automated complexity analysis of NUPRL extracts.* PhD thesis, Cornell University, 1999.

[4] G. Bonfante, JY Marion, and JY Moyen. The implicit complexity content of quasi-interpretation. In *Appsem'04*, 2004.

[5] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, January 1971.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[7] J-Y Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183, 2003.

[8] J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.

[9] Jean-Yves Moyen. *Analyse de la complexité et transformation de programmes.* Thèse d'université, Nancy 2, Dec 2003.