École Normale Supérieure

Langages de programmation et compilation

examen 2014–2015

Jean-Christophe Filliâtre

15 janvier 2015

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
Les deux problèmes sont indépendants. L'épreuve dure 3 heures.

---

1 Enregistrements

On considère ici une variante de mini-ML avec des entiers et des enregistrements, dont la syntaxe abstraite est la suivante :

e	::=	n	constante entière
	|	$-$	soustraction
	|	x	variable
	|	`let` $x = e$ `in` e	déclaration locale
	|	`ifzero` e `then` e `else` e	conditionnelle
	|	`fun` $x \to e$	abstraction
	|	$e\ e$	application
	|	$\{x = e; \ldots; x = e\}$	construction d'un enregistrement
	|	$e.x$	accès à un champ

Comme on l'imagine, la construction `ifzero` teste si la première expression est l'entier 0.

Question 1 Équiper ce langage d'une sémantique opérationnelle à petits pas traduisant une sémantique d'appel par valeur avec évaluation de la gauche vers la droite. Plus précisément, on définira la notion de valeur v, les réductions de tête $\xrightarrow{\epsilon}$ et les contextes de réduction E.

Question 2 Donner les étapes de réduction à petits pas de l'expression

```
(ifzero - 4 2 then fun x -> - x 1 else fun x -> - x 2) 44
```

ainsi que de l'expression

```
let id = fun x -> x in let r = { f = id id; g = id 42 } in r.g
```

Question 3 On souhaite typer ces programmes dans le système de Hindley-Milner, avec les types suivants

τ	::=	$\alpha \mid \tau \to \tau \mid$ `int` $\mid \{x = \tau; \ldots; x = \tau\}$	type
σ	::=	$\tau \mid \forall \alpha.\sigma$	schéma de type

où $\{x_1 : \tau_1; \ldots; x_n : \tau_n\}$ désigne le type (anonyme) d'un enregistrement ayant les champs $x_1, \ldots, x_n$, respectivement de types $\tau_1, \ldots, \tau_n$. Donner les règles de typage pour la soustraction (constante $-$), la construction `ifzero`, la construction d'un enregistrement et l'accès à un champ.

Question 4 Discuter la pertinence de types d'enregistrements anonymes dans un contexte où on chercherait à faire de l'inférence de types (par opposition à un langage comme OCaml où les types d'enregistrements doivent être déclarés préalablement).

Question 5 On adopte un schéma de compilation où tout enregistrement est un pointeur vers un bloc alloué sur le tas. Si un enregistrement contient n champs, ce bloc contient n mots et les champs y sont rangés par ordre alphabétique. Donner alors le code MIPS correspondant à la fonction suivante :

```
fun (p: { fst: int; snd: int }) ->
  ifzero p.fst then
    p
  else
    { snd = p.fst; fst = p.snd }
```

On supposera que l'argument `p` est passé dans le registre `$a0` et que le résultat de la fonction est renvoyé dans le registre `$v0`. Un aide-mémoire MIPS est donné à la fin du sujet.

Question 6 On souhaite maintenant ajouter le caractère mutable aux champs des enregistrements. Indiquer les modifications qui doivent être apportées à la syntaxe, à la sémantique, au typage et à la production de code. En ce qui concerne le typage, en particulier, on prendra soin de discuter le problème des références polymorphes (s'il est absent, indiquer pourquoi ; s'il est présent, expliquer comment y remédier).

---

2 Analyse de flot de données

Le contexte de ce problème est celui du langage RTL (*Register Transfer Language*). On va y réaliser une analyse statique, dans le but de réaliser des optimisations. On rappelle ici les différentes instructions du langage RTL :

i	::=	$r \leftarrow n$	chargement d'une constante
	|	$r \leftarrow r$	copie
	|	$r \leftarrow n(r)$	lecture en mémoire
	|	$n(r) \leftarrow r$	écriture en mémoire
	|	$r \leftarrow \mathit{unop}\ r$	opération unaire
	|	$r \leftarrow r\ \mathit{binop}\ r$	opération binaire
	|	$\mathit{ubranch}\ r$	branchement unaire
	|	$\mathit{bbranch}\ r\ r$	branchement binaire
	|	$r \leftarrow f(r, \ldots, r)$	appel de fonction
	|	$r \leftarrow \mathit{malloc}(n)$	allocation
	|	goto	saut inconditionnel

Les conventions sont les suivantes : n désigne une constante entière, r un pseudo-registre et L une étiquette de code (à laquelle on trouve une instruction RTL). La figure 1 contient le code RTL d'une fonction f qui reçoit un argument dans le pseudo-registre %1 et renvoie un résultat dans le pseudo-registre %2. Le point d'entrée est l'étiquette L_1 et le point de sortie l'étiquette L_{17}.

Question 7 Donner un programme C possible pour le code RTL de la figure 1.

```
%2 f(%1)
  entrée L1 sortie L17
L1 :  %3 ← 0            L2
L2 :  %4 ← %1           L3
L3 :  %5 ← 1            L4
L4 :  %6 ← %4 − %5      L5
L5 :  bgtz %6           L6, L16
L6 :  %7 ← %3           L7
L7 :  %8 ← %1           L8
L8 :  %9 ← 1            L9
L9 :  %10 ← %8 − %9     L10
L10 : %3 ← %7 + %10     L11
L11 : %11 ← %1          L12
L12 : %12 ← 1           L13
L13 : %13 ← %11 − %12   L14
L14 : %1 ← %13          L15
L15 : goto              L2
L16 : %2 ← %3           L17
```

FIGURE 1 – Code RTL d'une fonction f.

Expressions disponibles. Le but de notre analyse est de calculer, en chaque point du code RTL d'une fonction donnée, un ensemble d'*expressions disponibles*, c'est-à-dire de valeurs qui ont déjà été calculées et sont contenues dans des pseudo-registres. On représente de telles valeurs *symboliquement*, à l'aide de la syntaxe abstraite suivante :

v	::=	α_i	valeur arbitraire, inconnue
	|	n	constante entière
	|	$op\ v$	opération unaire
	|	$v\ op\ v$	opération binaire
	|	$v[n]$	accès en mémoire à l'adresse $v + n$

En particulier, la construction α_i nous permet de représenter une valeur arbitraire inconnue de l'analyse statique. C'est le cas notamment des arguments de la fonction. Ainsi, sur l'exemple de la figure 1, on donnera initialement au pseudo-registre %1 une valeur arbitraire α_1.

Plus précisément, on va calculer pour chaque étiquette L du graphe de flot de contrôle un ensemble $in(L)$ d'expressions disponibles avant l'exécution de l'instruction à l'étiquette L et un ensemble $out(L)$ d'expressions disponibles après son exécution. Sur l'exemple de l'instruction L_4 de la figure 1, on a les ensembles suivants

$$\begin{array}{rl} & in(L_4) = \{\alpha_2, \alpha_3, 1\} \\ L_4 : & \%6 \leftarrow \%4 - \%5 \quad L_5 \\ & out(L_4) = \{\alpha_2, \alpha_3, 1, \alpha_2 - 1\} \end{array}$$

où α_2 désigne la valeur contenue dans %1 et %4, et α_3 celle contenue dans %3.

Question 8 Donner des ensembles $in(L)$ et $out(L)$ possibles pour toutes les instructions de la figure 1 sous la forme d'un tableau :

étiquette L	$in(L)$	$out(L)$
L_1	$\{\alpha_1\}$	$\{\alpha_1, 0\}$
etc.		

Comme le code contient une boucle, on s'autorisera des approximations, en introduisant des valeurs arbitraires α_i aux endroits pertinents.

Calcul des expressions disponibles. Pour calculer les ensembles $in(L)$ et $out(L)$ de façon systématique, on va commencer par calculer les valeurs respectivement définies et invalidées par *une* instruction RTL. On notera respectivement $gen(L)$ et $kill(L)$ ces deux ensembles pour l'instruction située à l'étiquette L.

Question 9 Définir les ensembles $gen(L)$ et $kill(L)$ pour chaque instruction du langage RTL, en complétant le tableau suivant :

Instruction RTL	$gen(L)$	$kill(L)$
$r_1 \leftarrow r_2$ *binop* r_3	$\{e(r_2)$ *binop* $e(r_3)\}$	toute expression associée à r_1
etc.		

où $e(r)$ dénote la valeur de $in(L)$ contenue dans r, le cas échéant, et une nouvelle valeur α_i sinon.

Question 10 Donner des équations définissant $in(L)$ et $out(L)$

$$\begin{cases} in(L) &= \ldots \\ out(L) &= \ldots \end{cases}$$

en fonction de $gen(L)$, $kill(L)$, $in(L)$ et $out(L)$.

Question 11 Donner les types et profils de fonctions OCaml que vous écririez pour réaliser une telle analyse, en supposant donnés des types `Rtl.t` pour les instructions RTL, `Register.t` pour les pseudo-registres et `Label.t` pour les étiquettes de code. (On ne demande pas d'écrire le code OCaml, mais seulement de décrire son architecture.)

Sous-expressions communes. On souhaite maintenant exploiter le résultat de l'analyse des expressions disponibles pour effectuer l'optimisation dite des *sous-expressions communes*, qui consiste à éviter de faire plusieurs fois les mêmes calculs.

Question 12 Identifier les calculs redondants sur l'exemple de la figure 1, en montrant notamment comment l'analyse des expressions disponibles a permis de les trouver. Donner un code RTL simplifié en conséquence pour la fonction f.

Question 13 Donner un exemple de code RTL où une même expression est disponible en un certain point du programme, mais contenue dans deux pseudo-registres différents selon le chemin du graphe de flot de contrôle qui a permis d'atteindre ce point de programme.

Question 14 Décrire de manière générale comment l'analyse des expressions disponibles doit être utilisée pour réaliser l'optimisation des sous-expressions communes. On prendra soin d'expliquer comment le code mort résultant de cette optimisation est éliminé.

Annexe : aide-mémoire MIPS

On donne ici un fragment du jeu d'instructions MIPS. Vous êtes libre d'utiliser tout autre élément de l'assembleur MIPS. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`li`	r_1, n	charge la constante n dans le registre r_1
`la`	r_1, L	charge l'adresse de l'étiquette L dans le registre r_1
`addi`	r_1, r_2, n	calcule la somme de r_2 et n dans r_1
`add`	r_1, r_2, r_3	calcule la somme de r_2 et r_3 dans r_1 (on a de même `sub`, `mul` et `div`)
`move`	r_1, r_2	copie le registre r_2 dans le registre r_1
`lw`	r_1, n(`(`r_2`)`)	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`sw`	r_1, n(`(`r_2`)`)	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`beq`	r_1, r_2, L	saute à l'adresse désignée par l'étiquette L si $r_1 = r_2$ (on a de même `bne`, `blt`, `ble`, `bgt` et `bge`)
`beqz`	r_1, L	saute à l'adresse désignée par l'étiquette L si $r_1 = 0$ (on a de même `bnez`, `bltz`, `blez`, `bgtz` et `bgez`)
`j`	L	saute à l'adresse désignée par l'étiquette L
`jr`	r_1	saute à l'adresse contenue dans le registre r_1
`jal`	L	saute à l'adresse désignée par l'étiquette L, après avoir sauvegardé l'adresse de retour dans `$ra`
`jalr`	r_1	saute à l'adresse contenue dans le registre r_1, après avoir sauvegardé l'adresse de retour dans `$ra`

Quelques appels systèmes (`syscall`) :

appel	`$v0` (entrée)	`$a0` (entrée)	`$v0` (sortie)
`print_char`	11	caractère à afficher	
`print_int`	1	entier à afficher	
`print_string`	4	pointeur vers la chaîne à afficher	
`read_int`	5		entier lu
`sbrk` (`malloc`)	9	nombre d'octets à allouer	pointeur vers le bloc alloué