École Normale Supérieure

Langages de programmation et compilation

examen 2012–2013

Jean-Christophe Filliâtre

17 janvier 2013

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.

Les trois problèmes sont indépendants.

1 Démarrage

Question 1 Supposons que l'on veuille définir un nouveau langage L et écrire un compilateur pour ce langage écrit dans le langage L lui-même (c'est le cas des langages C, Java, OCaml, par exemple). Expliquer par quels procédés on peut obtenir un tel compilateur.

On se placera dans le cas réaliste d'un langage L comprenant de nombreuses constructions et où on cherche à obtenir un compilateur produisant du code efficace et qui soit lui-même efficace. On pourra introduire un certain nombre de compilateurs et/ou interprètes intermédiaires.

Question 2 La compilation croisée consiste à écrire un compilateur d'un langage L vers un langage machine N écrit dans le langage machine N alors qu'on ne dispose pas de la machine N mais seulement d'une machine M. On suppose que l'on dispose d'un compilateur C du langage L vers le langage machine M écrit en L et d'un compilateur D du langage L vers le langage machine N écrit en L.

Expliquer par quels procédés on peut réaliser la compilation croisée.

---

2 Lambda lifting

Dans ce problème, on considère un petit langage arithmétique où toutes les valeurs sont des entiers et où les expressions sont de la forme suivante :

e	::=	n	constante entière
	|	x	variable
	|	$e + e$	addition
	|	`let` $x = e$ `in` e	variable locale
	|	`ifzero` e `then` e `else` e	test à zéro
	|	$f(e, \ldots, e)$	appel de fonction
	|	`let rec` d `and` ... `and` d `in` e	fonctions locales
d	::=	$f(x, \ldots, x) = e$	définition de fonction

La construction `let rec` introduit un ensemble de fonctions mutuellement récursives locales, d étant la syntaxe abstraite d'une définition de fonction. Un programme est réduit à une expression. Voici un exemple de programme :

```
let rec fact(x) =
  let rec mult(y) = ifzero y then 0 else x + mult(y + -1) in
  ifzero x then 1 then mult(fact(x-1))
in
fact(10)
```

Question 3 Donner une sémantique opérationnelle à petits pas pour ce langage, qui traduise un appel par valeur avec évaluation de la gauche vers la droite, sous la forme d'un jugement $D, e \to D, e$ où D désigne un ensemble de définitions de fonctions d.

Lambda lifting. Pour compiler de tels programmes, on propose l'idée suivante : extraire toutes les définitions de fonctions locales pour en faire des définitions globales. L'idée est d'obtenir au final un programme de la forme

$$\begin{array}{l}\texttt{let rec } f(x,\ldots,x) = e' \\ \vdots \\ \texttt{and } f(x,\ldots,x) = e' \\ \texttt{in } e'\end{array}$$

où e' représente une expression qui ne contient plus la construction `let rec`. Sur l'exemple initial, on obtient le programme suivant :

```
let rec mult(x, y) = ifzero y then 0 else x + mult(x, y + -1)
and     fact(x)    = ifzero x then 1 then mult(x, fact(x-1))
in
fact(10)
```

Comme on le voit, on a dû ajouter un argument `x` à la fonction `mult`, car la variable `x` était libre dans le corps de la fonction `mult`. Cette idée s'appelle le *lambda lifting*.

Question 4 Donner un code MIPS pour le programme ci-dessus (après *lambda lifting*). Préciser la convention d'appel choisie.

Question 5 Donner le résultat du *lambda lifting* sur le programme suivant :

```
let rec foo(a, b) =
  let rec f(x) = ifzero x then 0 else a + g (x + -1)
  and     g(y) = ifzero y then 0 else b + f (y + -1)
  in
  f(10)
in
foo(1, 2)
```

Question 6 Donner un algorithme pour réaliser l'opération de *lambda lifting*. Expliquer la structure d'un programme OCaml qui le réaliserait (structures de données, découpage en fonctions, types des différentes fonctions). On ne demande pas d'écrire le code OCaml dans son intégralité.

On pourra supposer que, dans le programme initial, toutes les variables distinctes portent des noms distincts et que toutes les fonctions distinctes portent des noms distincts.

Question 7 La technique du *lambda lifting* ci-dessus s'applique-t-elle facilement à la compilation des programmes Pascal avec fonctions imbriquées (cours 7) ? Si oui, expliquer comment. Si non, expliquer pourquoi.

Question 8 La technique du *lambda lifting* ci-dessus s'applique-t-elle facilement à la compilation des programmes Mini-ML (cours 8) ? Si oui, expliquer comment. Si non, expliquer pourquoi.

3 Sous-expressions communes

Dans ce problème, on considère un petit langage arithmétique où toutes les valeurs sont des entiers et où les expressions sont de la forme suivante :

e	::=	n	constante entière
		x	variable
	|	$e + e$	addition
	|	`let` $x = e$ `in` e	variable locale
	|	`ifzero` e `then` e `else` e	test à zéro
	|	`print` e	affichage
	|	$f(e, \ldots, e)$	appel de fonction

L'expression `print` e affiche et renvoie la valeur de e. Un programme est un ensemble de fonctions mutuellement récursives globales et une expression servant de programme principal. Voici un exemple de tel programme :

```
def mult(x, y) = ifzero x then 0 else y + mult(x + -1, y)
def fact(x)    = ifzero x then 1 then mult(x, fact(x + -1))
print fact(10)
```

Question 9 Donner une sémantique opérationnelle à grands pas pour les expressions de ce langage, qui traduise un appel par valeur avec évaluation de la gauche vers la droite, sous la forme d'un jugement

$$e \Rightarrow v, L$$

où v est la valeur finale de l'expression e et L la liste ordonnée des valeurs qui ont été affichées par `print`.

Question 10 Donner un exemple de programme pour lequel l'expression e servant de programme principal n'admet pas de v et de L tels que $e \Rightarrow v, L$.

Sous-expressions communes. Pour compiler de tels programmes vers du code efficace, on propose l'idée suivante : si le corps d'une fonction fait apparaître deux fois la même expression e, on cherchera à ne l'évaluer qu'une seule fois, *quand cela est possible*, en introduisant un `let` x `=` e `in` ... On appelle cela le partage des sous-expressions communes.

Question 11 Réécrire le programme suivant avec cette idée :

```
def double(x) = x+x
def f(x) = print (double(x) + double(x))
print (f(1+2) + double(2) + double(2) + f(1+2))
```

Question 12 Même question avec le programme suivant :

```
def boucle(x) = boucle(x)
ifzero 1 then boucle(0)
         else ifzero 0 then 1 else boucle(0)
```

Quelle réflexion cela inspire-t-il ?

Question 13 On suppose donné un type OCaml `expr` pour la syntaxe abstraite des expressions. On dit qu'une expression est *pure* si son évaluation termine et n'implique jamais `print`. Écrire une fonction `is_pure: (string -> bool) -> expr -> bool` qui détermine si une expression donnée est pure. Le premier argument indique, pour chaque fonction f du programme, si le corps de f est une expression pure.

Question 14 En déduire une fonction OCaml qui détermine, pour chaque fonction f du programme, si le corps de f est une expression pure.

Question 15 Écrire une fonction OCaml `sec: expr -> expr` qui transforme une expression en y réalisant le partage des sous-expressions commune. On supposera qu'on a déjà déterminé les sous-expressions pures et que ce résultat est disponible sous la forme d'une fonction `pure: expr -> bool`.

Question 16 Indiquer où, dans le contexte du compilateur présenté aux cours 10 et 11, doit s'insérer l'optimisation des sous-expressions communes.

Question 17 Donner un autre exemple d'optimisation qu'un compilateur peut faire en exploitant l'analyse statique de pureté réalisée ci-dessus (questions 13 et 14).

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`)
`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`