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.

Correction : On note $X \xrightarrow{Y} Z$ un compilateur écrit dans le langage Y qui compile le langage source X vers le langage cible Z. Notre but est ici de construire $L \xrightarrow{L} M$, un compilateur de L vers M écrit en L, efficace et produisant du code efficace, mais aussi de le faire tourner sur la machine M, c'est-à-dire $L \xrightarrow{M} M$.

Soit L' un sous-ensemble du langage L. Soit L_0 un langage existant et $\mathrm{C}_0 : L_0 \xrightarrow{M} M$ un compilateur (ou interprète) de L_0 pas (nécessairement) efficace et produisant du code pas (nécessairement) efficace, pour la machine M.

On construit successivement

1. $\mathrm{C}_1 : L' \xrightarrow{M} M$ en écrivant un premier compilateur $L' \xrightarrow{L_0} M$ et en le compilant avec C_0 ; C_1 est inefficace et produit du code inefficace.
2. $\mathrm{C}_2 : L \xrightarrow{M} M$ en écrivant un second compilateur $L \xrightarrow{L'} M$ et en le compilant avec C_1 ; C_2 n'est pas efficace et il produit du code inefficace.
3. $\mathrm{C}_3 : L \xrightarrow{M} M$ en écrivant un troisième compilateur $L \xrightarrow{L} M$ et en le compilant avec C_2 ; C_3 est inefficace *mais* il produit du code efficace.
4. $\mathrm{C}_4 : L \xrightarrow{M} M$ en recompilant le troisième compilateur avec C_3 ; C_4 est efficace et il produit du code efficace.

Inutile de réaliser un autre compilateur $\mathrm{C}_5 : L \xrightarrow{M} M$ en utilisant C_4 car le point fixe est atteint.

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.

Correction : On suppose que par auto-compilation on construit un compilateur C' de L vers M qui s'exécute sur M (question précédente).
Soit le compilateur D de L dans N écrit dans L. On le compile à l'aide du compilateur C', on obtient un compilateur D_1 de L dans N qui s'exécute sur M. Ce compilateur sert à recompiler D pour obtenir un compilateur de L dans N qui s'exécute sur N.

---

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.

Correction : les valeurs sont les constantes entières

```
v ::= n
```

le jugement est `D, e -> D, e` où `D` est un ensemble de déclarations de fonctions
les réductions de tête sont

```
D, n1+n2 --> D, n  où n=n1+n2

D, let x = v1 in e2 --> D, e2[x <- v1]

D, ifzero 0 then e1 else e2 --> D, e1
```

```
D, ifzero v then e1 else e2 --> D, e2   si v<>0

D, f(v1,...,vn) --> D, e[xi <- vi]
     avec f(x1,...,xn) = e dans D

D, let rec d1 ... dn in e --> D d1...dn, e
```

les contextes de réductions sont définis par

```
E ::= []
    | E + e
    | v + E
    | let x = E in e
    | ifzero E then e else e
    | f(v,...,v,E,e,...e)
```

avec la règle

```
       D, e --> D', e'
    ---------------------
    D, E[e] --> D', E[e']
```

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.

Correction : conventions : arguments dans `a0` et `a1`, résultat dans `v0`. (note : `mult` renommé en `mult1`)

```
main:
        li      $a0, 10
        jal     fact
        move    $a0, $v0
        li      $v0, 1      # print
        syscall
        li      $v0, 10     # exit
        syscall

fact:   bnez    $a0, factE
        li      $v0, 1
        jr      $ra
factE:  addi    $sp, $sp, -8
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        addi    $a0, $a0, -1
        jal     fact
        move    $a1, $v0
        lw      $a0, 4($sp)
        lw      $ra, 0($sp)
        addi    $sp, $sp, 8
        j       mult1           # appel terminal !

mult1:  bnez    $a1, mult1E
        li      $v0, 0
        jr      $ra
mult1E: addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        addi    $a1, $a1, -1
        jal     mult1
        add     $v0, $a0, $v0
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
```

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)
```

Correction : on doit ajouter `a` à `f` (libre dans son corps) mais aussi `b` car `f` appelle `g`, qui a besoin de `b` ; idem pour `g`

```
let rec f(a, b, x) = ifzero x then 0 else a + g (a, b, x + -1)
```

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

on comprend qu'on va devoir faire un calcul de point fixe...

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.

Correction : D'une manière simple, on peut ajouter à la fonction `f` *toutes* les variables liées à l'endroit où `f` est définie (qu'elles soient liées par `let` ou comme paramètres de fonctions englobantes).

Cependant, on peut faire beaucoup mieux avec un calcul de point fixe (suggéré par la question précédente).

On se donne un module `S` pour représenter des ensembles de variables :

```
module S = Set.Make(String)
```

On se donne une table de hachage associant à toute fonction `f` l'ensemble de ses arguments supplémentaires :

```
let args: (string, S.t) Hashtbl.t = Hashtbl.create 17
```

On écrit une fonction `expr` qui calcule l'ensemble des variables libres d'une expression :

```
val expr: expr -> S.t
```

Cette fonction consulte la table `args` lorsqu'il y a des appels de fonctions, et la remplit lorsqu'il y a une définition de fonction. On fait alors un calcul de point fixe, à l'aide d'une référence booléenne qui est mise à `true` par la fonction `expr` dès lors que le contenu de la table `args` change :

```
let args = ... in
let change = ref true in
let expr e = ... in
while !change do
  change := false;
  ignore (expr pgm)
done;
args
```

où `pgm` désigne le programme (qui est une expression).

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.

Correction : Oui, mais il y a tout de même une subtilité : les variables étant modifiables en Pascal, il faut passer tous les arguments supplémentaires *par référence*.

On peut déterminer par une analyse préalable quelles sont les variables potentiellement modifiées ; on peut alors passer les autres par valeur, ce qui donnera un code un plus efficace.

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.

Correction : Non, car les fonctions sont des valeurs de première classe (elles peuvent être passées en arguments et renvoyées comme résultats d'autres fonctions) et les fonctions globales introduites par le *lambda lifting* ne nous permettent pas de représenter facilement les clôtures (il faudrait matérialiser une application partielle, mais c'est justement là le rôle d'une clôture).

On notera cependant que l'opération de *lambda lifting* correspond à l'opération de *closure conversion* : ce qui rend la compilation du langage ci-dessus plus simple que celle de Mini-ML, c'est que l'on sait toujours déterminer, au moment de l'appel, la valeur des variables qui ont été ajoutées (car on est toujours dans leur portée).

---

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`.

Correction :

```
 n --> n,[]

 e1 --> n1,L1  e2 --> n2,L2
 -------------------------
  e1+e2 --> n1+n2, L1L2

 e1 --> v1,L1  e2[x<-v1] --> v2,L2
 --------------------------------
     let x = e1 in e2 --> v2, L1L2

 e1 --> 0,L1  e2 --> v,L2                e1 --> n, L1  n<>0  e3 --> v,L3
 ------------------------------------    ------------------------------------
 ifzero e1 then e2 else e3 --> v, L1L2   ifzero e1 then e2 else e3 --> v, L1L3

  e --> v,L
-----------------
 print e --> v,Lv

  e1 --> v1,L1 ... en --> vn,Ln  f(x1,...,xn)=e   e[x1<-v1,...,xn<-vn] --> v,L
 ---------------------------------------------------------------------------
   f(e1,...,en) --> v, L1L2...LnL
```

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 $e \Rightarrow v, L$.

Correction :

```
def boucle(x) = boucle(x)
boucle(0)
```

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))
```

Correction :

```
def double(x) = x+x
def f(x) = print (let v = double(x) in v+v)
print (let x = 1+2 in
       let y = double(2) in
       f(x) + y + y + f(x))
```

On prend soin de ne pas partage `f(x)` car `f` fait un affichage.

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 ?

Correction : Ce programme ne doit pas être transformé en

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

sinon on modifie la sémantique (non terminaison). Ici, on ne peut effectuer aucun partage.

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.

Correction :

```
let rec is_pure fpure = function
  | Eprint _ -> false
  | Eint _ | Evar _ -> true
  | Eadd (e1, e2) | Elet (_, e1, e2) -> is_pure fpure e1 && is_pure fpure e2
  | Eif (e1, e2, e3) -> is_pure fpure e1 && is_pure fpure e2 && is_pure fpure e3
  | Ecall (f, el) -> fpure f && List.for_all (is_pure fpure) el
```

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

Correction : Calcul de point fixe avec la question précédente (car les fonctions sont mutuellement récursives).

```
let pure_funs fl =
  let h = Hashtbl.create 17 in
  List.iter (fun (f,_,_) -> Hashtbl.add h f false) fl;
  let change = ref true in
  let update (f, _, e) =
    if is_pure (Hashtbl.find h) e && not (Hashtbl.find h f) then begin
      Hashtbl.replace h f true; change := true
    end
  in
  while !change do change := false; List.iter update fl done;
  Hashtbl.find h
```

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`.

Correction : Cette question est difficile, surtout si on cherche à écrire un code relativement efficace. L'idée est d'écrire une fonction auxiliaire

```
val sec: map -> expr -> map * expr
```

où `map` est un dictionnaire associant à des expressions des noms de variables (le type `map` est `string M.t` où `M = Map.Make(E)` et `E = struct type t = expr ... end`).

```
let rec sec m e =
  let m, e = match e with
  | Evar _ | Eint _ -> m, e
  | Eadd (e1, e2) ->
      let m, e1 = sec m e1 in
      let m, e2 = sec m e2 in
      m, Eadd (e1, e2)
  | Eif (e1, e2, e3) ->
      let m, e1 = sec m e1 in
      let m2, e2 = sec m e2 in
      let m3, e3 = sec m e3 in
      let m, m2, m3 = inter_diff m2 m3 in
      m, Eif (e1, close m2 e2, close m3 e3)
  | Eprint e1 ->
      let m, e1 = sec m e1 in m, Eprint e1
  | Elet (x, e1, e2) ->
      let m, e1 = sec m e1 in
      let m2, e2 = sec m e2 in
      let m, m2x = filter x m2 in
      m, Elet (x, e1, close m2x e2)
  | Ecall (f, el) ->
      let add (m, el) e1 = let m, e1 = sec m e1 in m, e1 :: el in
      let m, el = List.fold_left add (m, []) el in
      m, Ecall (f, List.rev el)
  in
  match e with Evar _ | Eint _ -> m, e
    | _ -> if pure e then let m, v = add e m in m, Evar v else m, e
```

La fonction `add` cherche une entrée dans le dictionnaire et ajoute une nouvelle entrée si nécessaire, c'est-à-dire :

```
let var = let r = ref 0 in fun () -> incr r; string_of_int !r
let add e m =
  try let v = M.find e m in m, v
  with Not_found -> let v = var () in M.add e v m, v
```

La fonction `close` reconstruit une expression complète à partir d'un dictionnaire et d'une expression :

```
let make_let e1 x e2 = Elet (x, e1, e2)
let close = M.fold make_let
```

Les cas intéressants sont `if` et `let`. Pour le `if`, on partage tout ce que l'on peut entre `e1`, `e2` et `e3`, mais donc seulement ce qui est commun à `e2` et `e3` (sinon, on risque d'évaluer inutilement des sous-expressions de l'autre branche, et donc de changer dramatiquement la complexité du programme). D'où la fonction `inter_diff` (à écrire) qui calcule l'intersection $m_1 \cap m_2$ de deux dictionnaires m_1 et m_2 et les deux différences correspondantes ($m_1 \backslash m_2$ et $m_2 \backslash m_1$).
Pour le cas du `let x = e1 in e2`, il faut prendre soin de ne pas inclure dans le dictionnaire final des sous-expressions qui font intervenir la variable `x`, d'où l'utilisation d'une fonction `filter` (à écrire) qui partage un dictionnaire en deux dictionnaires selon que les entrées contiennent `x` ou non.
Note : la complexité de `sec` n'est pas linéaire, à cause de l'appel à `pure` qui est fait sur chaque sous-expression ; pour bien faire il suffirait que `sec` calcule elle-même la pureté, en même temps qu'elle réalise le partage des sous-expressions communes, c'est-à-dire qu'elle renvoie aussi un booléen indiquant la pureté.

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

Correction : Le plus simple est de l'insérer juste avant ou juste après la sélection d'instruction, à la manière de ce qui est suggéré dans les questions ci-dessus.
Cependant, il n'est pas impossible de le faire plus tard, par exemple sur le langage RTL, mais il faut alors réaliser une analyse de flot de données plus complexe. (Voir par exemple Appel.)

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).

Correction : Élimination de code mort, comme par exemple supprimer `e1` dans `let x = e1 in e2` si `e1` est pure et non utilisée dans `e2`.

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`