Master Informatique, M1 — 2012–2013
Introduction à la compilation — Durée : 3h
4 pages

Examen du 19 décembre 2012

Les notes de TD manuscrites ainsi que les transparents de cours et le polycopié de cette année sont les seuls documents autorisés. Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse.

1 Questions de cours

Question 1 Expliquer pourquoi le programme suivant est mal typé :

```
let g f = (f 1, f true) in g (fun x -> x)
```

Question 2 Dessiner la représentation mémoire des fermetures dans le programme suivant :

```
let rec g x y = if x = 0 then y else g (x - 1) y
let f x = let h = g x in fun z -> h 1 + h z
```

Question 3 Dessiner la représentation en mémoire des descripteurs des classes (avec les pointeurs vers les supers classes) et des objets suivants :

```
class A { int x; void f () { ... } }
class B extends A { int y; void g () { ... } }
class C extends A { void f () { ... }
                    void h () { ... } }
class D extends C { int z; void f () { ... } }
A o1 = new A()
A o2 = new B()
C o3 = new C()
A o4 = new D()
```

2 Compilation des exceptions

Dans ce problème, on considère un petit langage arithmétique avec exceptions, appelé Trap par la suite. Un programme Trap est une suite de définitions de fonctions introduites par `def`, mutuellement récursives, suivie d'un programme principal constitué d'une expression à afficher introduite par `print`. Chaque fonction a un unique argument entier et un corps qui est une expression entière. Les expressions sont formées à partir de constantes entières, de variables, des quatre opérations arithmétiques, d'une conditionnelle de la forme `ifzero then else`, d'une construction `let in` pour introduire une variable locale, d'une construction `raise` pour lever une exception et d'une construction `try with` pour la rattraper. Voici deux exemples de programmes Trap (qu'il n'est pas nécessaire de comprendre) :

```
def f(x) =
  ifzero x-1 then raise(0) else x/2
def syra(n) =
  let m = f(n) in
  ifzero n - 2*m then syra(m)
  else syra(3*n+1)
print
  try syra(42) with x -> x
```

```
def search(x) =
  ifzero x then 0 else
  ifzero x/17 then raise(1) else
  try  1 + search(x-17)
  with y -> ifzero x/42 then raise(y+1) else
            try 1 + search(x-42) with z -> raise(y+z)
print
  try search(123) with e -> 0-e
```

La sémantique de TRAP est identique à celle de OCaml, à ces trois points près :

- il y a une unique exception, contenant une valeur entière, que l'on aurait par exemple déclarée en OCaml avec `exception E of int` ; la construction `raise(`e`)` correspondrait alors en OCaml à `raise (E` e`)`, et la construction `try` e_1 `with` x `->` e_2 correspondrait à `try` e_1 `with E` x `->` e_2 ;
- la construction `ifzero` e_1 `then` e_2 `else` e_3 s'écrirait en OCaml `if` e_1`=0 then` e_2 `else` e_3 ;
- l'évaluation se fait de la gauche vers la droite ; ainsi, si par exemple l'évaluation de e_1 lève une exception, alors l'évaluation de e_1 `+` e_2 lèvera cette exception et e_2 ne sera pas évaluée.

2.1 Analyse des fonctions pouvant lever une exception

Dans cette partie, on souhaite déterminer si l'évaluation d'un programme peut conduire à une exception non rattrapée (par exemple pour le rejeter). Comme il s'agit d'une propriété non décidable de manière générale, on va se contenter d'une condition suffisante. L'idée est de déterminer *pour chaque fonction* si son évaluation est susceptible de lever une exception non rattrapée. On peut alors déterminer ce qu'il en est de l'expression qui constitue le programme principal.

Question 4 Pour le programme suivant, déterminer quelles sont les fonctions dont l'évaluation peut conduire à une exception non rattrapée (on rappelle que les fonctions sont mutuellement récursives).

```
def f(x) = raise(x)
def g(x) = ifzero x then f(x+1) else i(x)
def h(x) = ifzero x then 1 else x * h(x-1)
def i(x) = try 1 + g(x) with y -> h(y)
def j(x) = x + try g(x) with y -> f(y)
print ...
```

On se donne les types OCaml suivants pour représenter la syntaxe abstraite de TRAP. (La nature des variables n'est pas importante dans cette partie.)

```
type binop = Add | Sub | Mul | Div
type var = ...
type expr =
  | Const of int
  | Var of var
  | Binop of binop * expr * expr
  | Letin of var * expr * expr
  | Ifzero of expr * expr * expr
  | Call of string * expr
  | Raise of expr
  | TryWith of expr * var * expr
type def = { name : string; arg : var; body : expr; }
type program = { funs : def list; print : expr; }
```

Question 5 Écrire une fonction `expr : (string -> bool) -> expr -> bool` qui détermine si l'évaluation d'une expression est susceptible de lever une exception non rattrapée, le premier argument indiquant pour chaque fonction du programme si son évaluation peut conduire à une exception non rattrapée.

Pour déterminer si l'évaluation d'un programme est susceptible de lever une exception non rattrapée, il faut maintenant calculer la fonction de type `string -> bool`, appelée `raise_exn` par la suite, passée comme premier argument à la fonction `expr` ci-dessus.

Question 6 On aimerait simplement pouvoir définir la fonction `raise_exn` par le point fixe suivant

```
let rec raise_exn f = (expr raise_exn (body f)) = raise_exn f)
```

où `body : string -> expr` est supposée renvoyer le corps des fonctions. Malheureusement, cette définition de fonction ne convient pas. Justifier cette affirmation.

Question 7 Pour résoudre le problème, on propose de calculer ce point fixe de manière itérative. Compléter le code de la fonction `fixpoint : program -> (string -> bool)` ci-dessous pour calculer `raise_exn` :

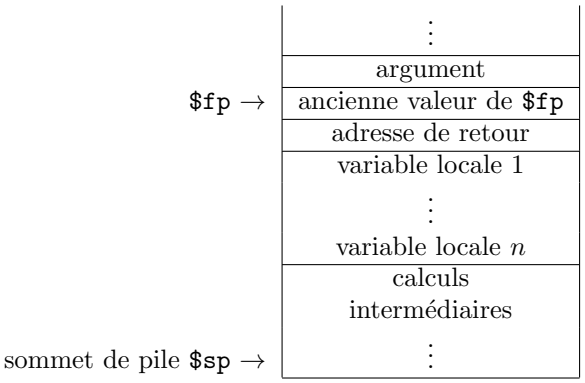
```
let fixpoint p =
  let r = Hashtbl.create 17 in
  let raise_exn f = Hashtbl.find r f in
  List.iter (fun f -> Hashtbl.add r f.name false) p.funs;
  let rec fix () =  (* à compléter *)  in
  fix ();
  raise_exn
```

2.2 Production de code

On s'intéresse enfin à la compilation de TRAP vers l'assembleur MIPS (un aide-mémoire MIPS est donné à la fin de ce sujet). On considère un schéma de compilation simple utilisant la pile, sans chercher à faire d'allocation de registres. On se donne les conventions d'appel suivantes :

- l'argument est passé sur la pile, et empilé et dépilé par l'appelant ;
- le tableau d'activation est créé et détruit par l'appelé, son adresse la plus haute est désignée par `$fp` et il contient, de haut en bas :
  - l'ancienne valeur de `$fp`,
  - l'adresse de retour,
  - les variables locales impliquées dans le corps de la fonction.

On notera que les tableaux d'activation ne sont pas contiguës, car séparés par d'éventuels calculs intermédiaires stockés sur la pile. Le schéma suivant illustre le tableau d'activation le plus récent :



On rappelle que la pile croît vers le bas, *i.e.* vers des adresses de plus en plus petites.

On suppose que l'analyse de portée a été réalisée et que chaque variable est directement représentée par un entier qui désigne sa position par rapport à `$fp`. La compilation s'effectue donc sur la syntaxe abstraite présentée dans la partie précédente, avec

```
type var = int
```

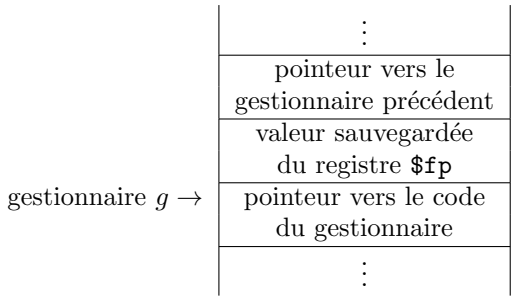
L'argument d'une fonction est donc représenté par l'entier 4 et les variables locales introduites par les constructions `let` et `try` par les entiers -8, -12, etc.

Question 8 On suppose que le code MIPS d'une expression est produit par une fonction OCaml `compile_expr : expr -> mips`. Le code MIPS produit a pour effet de stocker la valeur de l'expression dans le registre `$v0`. Donner le code de cette fonction correspondant aux constructeurs `Var`, `Letin` et `Call` de la syntaxe abstraite. On s'autorisera à écrire librement du code MIPS au sein du code OCaml.

Question 9 Définir une fonction OCaml `compile_def : def -> mips` produisant le code d'une fonction de TRAP, en supposant que le type `def` contient, outre le nom et le corps de la fonction, le nombre de variables locales à allouer (contenu dans le champ `locals`) :

```
type def = { name : string; body : expr; locals : int; }
```

Pour compiler les exceptions, on va utiliser la technique dite du *stack cutting*. Chaque construction `try` va déposer sur la pile un *gestionnaire* d'exception, constitué de trois éléments : un pointeur vers le gestionnaire précédent ; la valeur du registre `$fp` ; et un pointeur vers le code du gestionnaire (l'expression qui suit le `with`). Un gestionnaire g est désigné par l'adresse la plus basse de ces trois éléments. On a donc la situation suivante :



On choisit de conserver le gestionnaire le plus récent dans le registre MIPS `$v1`. La valeur contenue dans l'exception sera, le cas échéant, stockée dans le registre `$v0`.

Question 10 Dessiner l'état de la pile lorsque l'expression `raise(x)` du corps de la fonction `f` ci-desous est exécutée par le programme suivant (on demande seulement de dessiner les gestionnaires d'exception). Justifier ainsi la présence sur la pile des trois éléments constituant le gestionnaire d'exception.

```
def f(x) = try raise(x) with y -> raise (y+1)
def g(x) = try f(x) with y -> raise (y+2)
print g(1)
```

Question 11 Donner le code de la fonction `compile_expr` pour les constructeurs `Raise` et `TryWith`.

Annexe : aide-mémoire MIPS

On donne ici un fragment du jeu d'instructions MIPS suffisant pour réaliser ce problème. (Vous êtes cependant libre d'utiliser tout autre élément de l'assembleur MIPS.) Les registres utilisés ici, et leur signification dans le schéma de compilation adopté, sont les suivants :

`$v0`	résultat d'une expression
`$fp`	pointeur sur le tableau d'activation courant
`$sp`	pointeur sur le sommet de la pile
`$v1`	pointeur sur le gestionnaire courant
`$ra`	adresse de retour
`$a0`	registre libre d'usage

Les instructions susceptibles d'être utiles sont les suivantes (où les r_i désignent des registres, n une constante entière et L une étiquette de code) :

`addi` r_1`,` r_2`,` n	calcule la somme de r_2 et n dans r_1
`move` r_1`,` r_2	copie le registre r_2 dans le registre r_1
`la` r`,` L	charge l'adresse désignée par l'étiquette L dans le registre r
`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
`j` L	saute à l'adresse désignée par l'étiquette L
`jr` r	saute à l'adresse contenue dans le registre r
`jal` L	saute à l'adresse désignée par l'étiquette L, après avoir sauvegardé l'adresse de retour dans `$ra`