École Normale Supérieure

Langages de programmation et compilation

examen 2008–2009

Jean-Christophe Filliâtre

22 janvier 2009

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

Dans ce problème, on considère un petit langage arithmétique avec exceptions, appelé $\mathcal{L}$ par la suite. Un programme $\mathcal{L}$ 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 $\mathcal{L}$ (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 $\mathcal{L}$ est identique à celle de Caml, à ces trois points près :

- il y a une unique exception, contenant une valeur entière, que l'on aurait par exemple déclarée en Caml avec `exception E of int` ; la construction `raise(`e`)` correspondrait alors en Caml 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 Caml `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.

1 Sémantique

On se donne la syntaxe abstraite suivante pour les expressions de $\mathcal{L}$:

e	::=	n	constante entière
	|	x	variable
	|	e *op* e	opération arithmétique $op \in \{+, -, \times, /\}$
	|	`let` $x = e$ `in` e	variable locale
	|	`ifzero` e `then` e `else` e	conditionnelle
	|	$f(e)$	application
	|	`raise`(e)	levée d'exception
	|	`try` e `with` $x \to e$	gestionnaire d'exception

On souhaite donner au langage $\mathcal{L}$ une sémantique opérationnelle à petits pas, où la notion de valeur est la suivante :

v	::=	n	constante entière
	|	`raise`(n)	exception, contenant la constante entière n

Autrement dit, une valeur est soit une constante entière, soit une exception contenant une constante entière.

Question 1 Donner la suite de réductions élémentaires de l'expression `try (1+2)+raise(4) with x -> raise(x+1)`.

Correction :

```
    try (1+2)+raise(4) with x -> raise(x+1)
--> try 3+raise(4) with x -> raise(x+1)
--> try raise(4) with x -> raise(x+1)
--> raise(4+1)
--> raise(5)
```

Question 2 Donner les règles définissant la sémantique de $\mathcal{L}$.

Correction : les réductions de tête sont

$$\begin{array}{rcll}
n_1 \; op \; n_2 & \xrightarrow{\epsilon} & n & \text{avec } n = op(n_1, n_2) \\
\texttt{let } x = n \texttt{ in } e & \xrightarrow{\epsilon} & e[x \leftarrow n] & \\
f(n) & \xrightarrow{\epsilon} & e[x \leftarrow n] & \text{avec } f(x) = e \\
\texttt{ifzero } 0 \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\epsilon} & e_1 & \\
\texttt{ifzero } n \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\epsilon} & e_2 & \text{avec } n \neq 0 \\
\texttt{try } n \texttt{ with } x \to e & \xrightarrow{\epsilon} & n & \\
\texttt{try raise}(n) \texttt{ with } x \to e & \xrightarrow{\epsilon} & e[x \leftarrow n] & \\
F(\texttt{raise}(n)) & \xrightarrow{\epsilon} & \texttt{raise}(n) & \text{avec } F \neq \square
\end{array}$$

et les contextes sont

$$\begin{array}{rcl}
E & ::= & \square \\
& | & E \; op \; e \\
& | & n \; op \; E \\
& | & \texttt{let } x = E \texttt{ in } e \\
& | & \texttt{ifzero } E \texttt{ then } e \texttt{ else } e \\
& | & f(E) \\
& | & \texttt{raise}(E) \\
& | & \texttt{try } E \texttt{ with } x \to e
\end{array}
\qquad
\begin{array}{rcl}
F & ::= & \square \\
& | & F \; op \; e \\
& | & n \; op \; F \\
& | & \texttt{let } x = F \texttt{ in } e \\
& | & \texttt{ifzero } F \texttt{ then } e \texttt{ else } e \\
& | & f(F) \\
& | & \texttt{raise}(F)
\end{array}$$

la réduction en un pas

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \to E(e_2)}$$

---

2 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 3 Pour le programme suivant, déterminer quelles sont les fonctions dont l'évaluation peut conduire à une exception non rattrapée.

```
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 rappelle que les fonctions sont mutuellement récursives.

Correction : `f`, `g` et `j`

On se donne les types Caml suivants pour représenter la syntaxe abstraite de $\mathcal{L}$. (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 4 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.

Correction :

```
let rec expr h = function
  | Cst _ | Var _ -> false
  | Binop (_, e1, e2)
  | Letin (_, e1, e2) -> expr h e1 || expr h e2
  | If (e1, e2, e3) -> expr h e1 || expr h e2 || expr h e3
  | Call (f, e) -> h f || expr h e
  | Raise _ -> true
  | TryWith (e1, _, e2) -> expr h e1 && expr h e2
```

Question 5 En déduire une fonction `check : program -> bool` qui détermine si l'évaluation d'un programme est susceptible de lever une exception non rattrapée.

Correction :

```
let check p =
  let raises = Hashtbl.create 17 in
  let get f = Hashtbl.find raises f in
  List.iter (fun f -> Hashtbl.add raises f.name false) p.funs;
  let rec fixpoint () =
    let fixpoint_reached = ref true in
    List.iter
      (fun f ->
 if not (get f.name) && expr get f.body then begin
   fixpoint_reached := false;
   Hashtbl.add raises f.name true
 end)
      p.funs;
    if not !fixpoint_reached then fixpoint ()
  in
  fixpoint ();
  expr get p.print
```

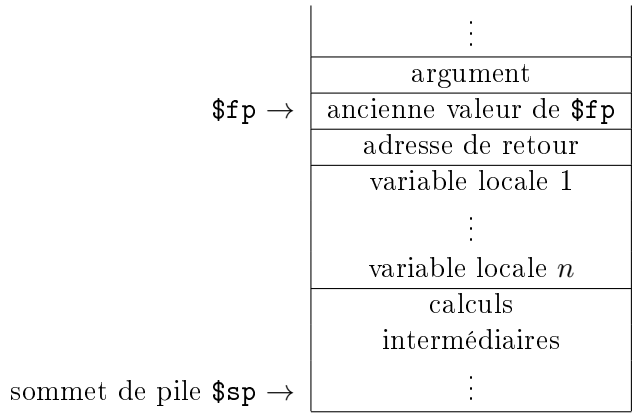
---

3 Production de code

On s'intéresse enfin à la compilation de $\mathcal{L}$ 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 6 On suppose que le code MIPS d'une expression est produit par une fonction Caml `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 Caml.

Correction :

```
let rec compile_expr = function
  | Var fp ->
      lw  $v0,  fp($fp)
  | Letin(fp, e1, e2) ->
      compile_expr e1
      sw  $v0,  fp($fp)
      compile_expr e2
  | Call (f, e) ->
      sub $sp, $sp, 4
      compile_expr e
      sw  $v0, 0($sp)
      jal f
      add  $sp, $sp, 4
```

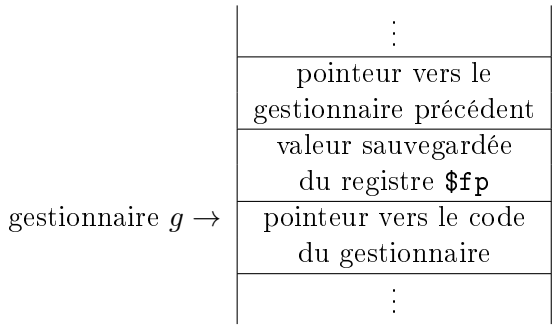
Question 7 Définir une fonction Caml `compile_def : def -> mips` produisant le code d'une fonction de $\mathcal{L}$, en supposant que le type `def` contient, outre le nom et le corps de la fonction, le nombre de variables locales à allouer (contenu dans le champ `locals`) :

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

Correction :

```
let compile_def f =
  f.name:
     sub $sp, $sp, f.locals
     sw  $fp, (f.locals - 4)($sp)
     add $fp, $sp, (f.locals - 4)
     sw  $ra, -4($fp)
     compile_expr f.body
     lw  $ra, -4($ fp)
     lw  $fp, 0($fp)
     add $sp, $sp, f.locals
     jr  $ra
```

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 8 Justifier la présence sur la pile des trois éléments constituant le gestionnaire d'exception.

Correction : La valeur de `$fp` est nécessaire pour retrouver le contexte du `try with`, qui peut être différent (si l'exception est levée à l'intérieur d'une fonction appelée) ; le pointeur vers le gestionnaire précédent est nécessaire pour rétablir le `try with` englobant (les gestionnaires peuvent être les uns dans les autres) ; enfin le code du gestionnaire est évidemment nécessaire.

Question 9 Donner le code de la fonction `compile_expr` correspondant aux constructeurs `Raise` et `TryWith`.

Correction :

```
| Raise e ->
    compile_expr e
    move $sp, $v1
    lw   $v1, 8($sp)
    lw   $fp, 4($sp)
    lw   $a0, 0($sp);
    add  $sp, $sp, 12
    jr   a0
| TryWith (e1, x, e2) ->
    let handler = label () in
    let done_ = label () in
    sub  $sp, $sp, 12
    sw   $v1, 8($sp)
    Move $v1, $sp
    sw   $fp, 4($sp)
    la   $a0, handler
    sw   $a0, 0($sp)
    compile_expr e1
    lw   $v1, 8($sp)
    add  $sp, $sp, 12
    b    done_;
handler:
    sw   $v0, x($fp)
    compile_expr e2
done_:
```

Question 10 Si on souhaite réaliser une production de code efficace avec allocation de registres, quelle serait la difficulté posée par ce schéma de compilation des exceptions ? On pourra considérer le cas des registres *callee-saved* dans la compilation d'une expression telle que

$$e_1 + \texttt{try}\ f(e_2)\ \texttt{with}\ x \to e_3$$

Donner une solution simple pour résoudre ce problème. Quelle(s) solution(s) plus efficace(s) pourriez-vous proposer ?

Correction : Avec ce schéma de compilation, les registres *callee-saved* ne sont pas restaurés lors du déclenchement d'une exception. Dans l'exemple donné, si f lève une exception, alors il faut restaurer les registres *callee-saved* avant de revenir exécuter l'expression e_3.

Une solution simple : ne pas utiliser de registre *callee-saved* ! (fonctionne très bien en pratique)

Une autre solution : on dépile successivement tous les tableaux d'activation, en restaurant les registres *callee-saved* au passage, jusqu'à tomber sur le gestionnaire ; c'est la technique du *stack unwinding*.

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`