École Normale Supérieure

Langages de programmation et compilation

examen 2011–2012

Jean-Christophe Filliâtre

19 janvier 2012

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

Les deux problèmes sont indépendants.

1 Interprétation abstraite

Dans ce problème, on considère un petit langage arithmétique très simple, appelé $\mathcal{L}$ par la suite. Un programme $\mathcal{L}$ est une suite de définitions de fonctions introduites par `def` et mutuellement récursives. 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 l'argument de la fonction, des quatre opérations arithmétiques, d'une conditionnelle de la forme `ifzero then else` et d'appels de fonctions. On suppose que les entiers sont de précision arbitraire. Voici un exemple de programme :

```
def power2(x) =
  ifzero x then 1 else 2 * power2(x-1)
def f(x) =
  zero(x) - power2(x)
def zero(x) =
  ifzero x then 0 else zero(x-1)
```

On s'intéresse ici à déterminer le signe de chaque expression d'un programme. Comme il s'agit d'une propriété non décidable de manière générale, on va se contenter d'une approximation. L'idée est de déterminer pour chaque fonction le "signe" possible de la valeur qu'elle renvoie. Un signe s peut prendre ici cinq valeurs différentes, données par le type OCaml suivant :

```
type sign = Bot | Neg | Zero | Pos | Top
```

Un signe s est interprété comme une partie de $\mathbb{Z}$, notée $I(s)$ et définie de la façon suivante :

$$\begin{aligned} I(\texttt{Bot}) &= \emptyset \\ I(\texttt{Neg}) &= \{i \in \mathbb{Z} \mid i < 0\} \\ I(\texttt{Zero}) &= \{0\} \\ I(\texttt{Pos}) &= \{i \in \mathbb{Z} \mid i > 0\} \\ I(\texttt{Top}) &= \mathbb{Z} \end{aligned}$$

On interprète alors le jugement « l'expression e a le signe s » comme « la valeur de e appartient nécessairement à l'ensemble $I(s)$ » et le jugement « la fonction f a le signe s » comme « quel que soit x, la valeur de $f(x)$, si elle existe, appartient nécessairement à $I(s)$ ».

Question 1 Pour le programme donné plus haut en exemple, déterminer le signe de chacune des trois fonctions `power2`, `f` et `zero`.

Question 2 Écrire une fonction `sign_add: sign -> sign -> sign` qui, étant donnés les signes de x et y, donne le signe de l'expression $x+y$. Écrire de même une fonction `sign_sub` pour la soustraction. (Dans la suite, on supposera avoir écrit également des fonctions `sign_mul` et `sign_div`.)

On se donne les types OCaml suivants pour représenter la syntaxe abstraite de $\mathcal{L}$. (La nature des variables n'est pas importante.)

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

Question 3 Écrire une fonction `expr: (string -> sign) -> expr -> sign` qui détermine le signe d'une expression, le premier argument donnant le signe de chaque fonction du programme.

Question 4 Écrire une fonction qui prend un programme (de type `program`) en argument et renvoie une table donnant le signe de chacune de ses fonctions. (On pourra réaliser la table par la méthode de son choix.)

Question 5 Donner un exemple d'optimisation qu'un compilateur peut effectuer en exploitant le résultat d'une telle analyse.

Question 6 Expliquer comment modifier la méthode ci-dessus (questions 3 et 4) pour prendre en compte le signe de l'argument de chaque fonction.

Question 7 Donner un exemple où la prise en compte du signe de l'argument donne un résultat différent de celui donné par la méthode proposée initialement.

---

2 Évaluation paresseuse

Dans ce problème, on considère le langage mini-ML dont la syntaxe abstraite est la suivante :

e	::=	x	variable
	|	n	constante entière
	|	op	primitive ($+$, $-$, $\times$, $/$, `ifz`)
	|	`fun` $x \to e$	fonction
	|	$e\ e$	application
	|	`let` $x = e$ `in` e	liaison locale

Les valeurs sont ici limitées aux entiers et aux fonctions. Il y a cinq primitives : les quatre opérations arithmétiques et un opérateur de test, `ifz`. L'expression `ifz` e_1 e_2 e_3 attend une expression entière e_1 et deux expressions e_2 et e_3 de même type ; elle s'évalue en e_2 si la valeur de e_1 est 0 et en e_3 sinon.

Typage. On munit ce langage de types simples, de la forme

$$\tau \quad ::= \quad \texttt{int} \mid \tau \to \tau$$

Un environnement Γ associe un type à chaque variable x, noté $\Gamma(x)$. On note $\Gamma \vdash e : \tau$ le jugement « dans l'environnement Γ, l'expression e a le type τ ».

Question 8 Donner des règles d'inférence pour le jugement $\Gamma \vdash e : \tau$. (On ne demande pas un *algorithme* de typage.)

Sémantique. En cours, nous avions muni un tel langage d'une sémantique en appel par valeur (cours 3). Ici, nous considérons une autre sémantique : l'*évaluation paresseuse*. L'idée est la suivante : pendant l'évaluation d'une expression de la forme $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ ou de la forme $e_2\ e_1$, la sous-expression e_1 n'est évaluée que si sa valeur s'avère nécessaire. Plus formellement, on se donne une sémantique opérationnelle à petits pas de la manière suivante. La réduction en tête $\xrightarrow{\epsilon}$ est définie par

$$\begin{array}{rcll}
(\texttt{fun}\ x \to e_2)\ e_1 & \xrightarrow{\epsilon} & e_2[x \leftarrow e_1] & \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \xrightarrow{\epsilon} & e_2[x \leftarrow e_1] & \\
\texttt{ifz}\ 0\ e_2\ e_3 & \xrightarrow{\epsilon} & e_2 & \\
\texttt{ifz}\ n\ e_2\ e_3 & \xrightarrow{\epsilon} & e_3 & \text{si } n \neq 0 \\
op\ n_1\ n_2 & \xrightarrow{\epsilon} & n & \text{avec } op \in \{+, -, \times, /\} \text{ et } n = n_1\ op\ n_2
\end{array}$$

La réduction en un pas est alors définie par la notion suivante de contexte :

$$\begin{array}{rcll}
E & ::= & \square & \\
& \mid & op\ E\ e & \text{avec } op \in \{+, -, \times, /\} \\
& \mid & op\ n\ E & \text{avec } op \in \{+, -, \times, /\} \\
& \mid & \texttt{ifz}\ E\ e\ e &
\end{array}$$

Dit autrement, les seules expressions qui forcent l'évaluation sont les quatre opérations arithmétiques, qui évaluent leurs deux arguments, et l'opérateur `ifz` qui évalue son premier argument. On note qu'à chaque fois l'évaluation n'est forcée que lorsque la primitive est totalement appliquée.

Question 9 Donner toutes les étapes de l'évaluation de l'expression suivante :

```
let x = (+ 1) 2 in
let y = (- 3) 4 in
((ifz y) x) y
```

Question 10 Donner un exemple de programme dont l'évaluation diffère, au final, de celle en appel par valeur. (On pourra considérer qu'une division par zéro provoque un arrêt de l'évaluation.)

Compilation. Une manière simple de compiler ce langage en respectant la sémantique ci-dessus consiste à remplacer une expression e qu'on ne souhaite pas évaluer tout de suite par la fonction $\texttt{fun}\ \_ \to e$, puis à appliquer cette fonction plus tard lorsqu'on souhaite finalement obtenir la valeur de e. On peut alors compiler le programme obtenu comme on le fait en appel par valeur. C'est cependant une manière inefficace de procéder, car une même expression pourra être évaluée plusieurs fois. Ainsi, dans l'exemple de la question 9 ci-dessus, on évaluerait l'expression `(- 3) 4` deux fois. Aussi, on adopte un schéma de compilation plus subtil. Une expression e dont on souhaite différer l'exécution est représentée par un *pointeur* p vers la fermeture représentant $\texttt{fun}\ \_ \to e$; on appelle cela un *glaçon*. Si la valeur d'un tel glaçon est nécessaire, on la calcule en appliquant la fermeture

puis on modifie le pointeur p pour qu'il pointe désormais vers cette valeur. Ainsi, la prochaine fois que la valeur sera exigée, elle sera directement disponible et ne sera pas recalculée. On parle alors de *glaçon dégelé*.

En pratique, on procède ainsi. On commence par effectuer une transformation de programme, notée T, qui introduit une construction explicite des glaçons, notée `G`.

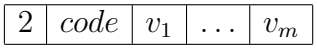
$$\begin{array}{rcl} T(\texttt{fun}\ x \to e) & = & \texttt{fun}\ x \to T(e) \\ T(e_1\ e_2) & = & T(e_1)\ (\texttt{G}\ (\texttt{fun}\ \_ \to T(e_2))) \\ T(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \texttt{let}\ x = \texttt{G}\ (\texttt{fun}\ \_ \to T(e_1))\ \texttt{in}\ T(e_2) \\ T(e) & = & e \quad \text{sinon} \end{array}$$

On compile alors le programme obtenu par cette transformation comme cela a été vu en cours, en adoptant la représentation suivante des valeurs. Toute valeur est un pointeur vers un *bloc* alloué sur le tas, formé de $t+1$ mots consécutifs. Le premier de ces mots contient un entier qui dénote la nature de la valeur et on l'appelle l'*étiquette*. L'entier $t \geq 0$ est appelé la *taille* du bloc. Les différents types de valeurs sont les suivants :

— Une constante entière n est un bloc d'étiquette 0 et de taille 1

où le second mot contient la valeur de n.

— Une fermeture est un bloc d'étiquette 2 et de taille $m+1$



où le second mot contient le pointeur *code* vers le code à exécuter et les mots suivant contiennent l'environnement, sous la forme de m valeurs (cf cours 8). Le code d'une fermeture suppose que l'argument de la fonction est contenu dans le registre `$a0` et la fermeture elle-même dans le registre `$a1`, et renvoie son résultat dans le registre `$v0`.

— Un glaçon est un bloc d'étiquette 3 et de taille 1

où f est une fermeture (c'est-à-dire un pointeur vers un bloc du type précédent), dont l'argument n'est pas significatif. C'est la construction `G` qui construit un tel bloc.

— Un glaçon dégelé, *i.e.* dont on a déjà calculé la valeur v, est un bloc d'étiquette 4 et de taille 1, de la forme

Un tel bloc est obtenu par modification en place d'une valeur du type précédent (glaçon). On garantira par la suite que la valeur v d'un glaçon dégelé n'est jamais un glaçon (dégelé ou pas), c'est-à-dire a une étiquette inférieure ou égale à 2.

Toute la subtilité de l'évaluation paresseuse tient dans une fonction `force` qui prend en argument une valeur v et force son évaluation. Le pseudo-code de `force` est le suivant :

```
force(v) =def
  e ← premier champ de v (son étiquette)
  si e ≤ 2 renvoyer v
  si e = 4 renvoyer le second champ de v
  sinon (v est un glaçon)
    f ← second champ de v (c'est une fermeture)
    w ← appel de f
    w ← force(w)
    premier champ de v ← 4
    second champ de v ← w
    renvoyer w
```

Question 11 Écrire le code MIPS de la fonction `force`. On suppose que l'argument v est passé dans le registre `$a0` et que le résultat est renvoyé dans ce même registre `$a0` (afin de simplifier l'utilisation de `force`). Un aide-mémoire MIPS est donné à la fin du sujet.

Question 12 Écrire le code MIPS correspondant à la primitive $+$. On supposera que ses deux arguments sont contenus dans les registres `$a0` et `$a1`.

Listes et filtrage. On étend le langage avec des listes. Les expressions sont étendues, d'une part avec deux nouvelles primitives `[]` et :: pour la construction des listes, et d'autre part avec une construction de filtrage :

$$e \quad ::= \quad ... \mid \texttt{match}\ e\ \texttt{with}\ \texttt{[]} \to e \mid x :: x \to e$$

Question 13 Donner les nouvelles règles de typage, en supposant que les types sont étendus de la manière suivante :

$$\tau \quad ::= \quad ... \mid \tau\ \texttt{list}$$

Question 14 On souhaite que l'évaluation de l'expression $\texttt{match}\ e_1\ \texttt{with}\ \texttt{[]} \to e_2 \mid x :: y \to e_3$ force l'évaluation de l'expression e_1, mais uniquement pour déterminer si sa valeur est de la forme `[]` ou ::. Étendre la sémantique opérationnelle en conséquence *i.e.* donner de nouvelles règles pour $\xrightarrow{\epsilon}$ et étendre la notion de contexte E.

Récursivité. On ajoute au langage des définitions récursives, c'est-à-dire une nouvelle construction

$$e \quad ::= \quad ... \mid \texttt{let rec}\ x = e\ \texttt{in}\ e$$

avec la règle de typage

$$\frac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

et la règle de réduction

$$\texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 \quad \xrightarrow{\epsilon} \quad e_2[x \leftarrow \texttt{let rec}\ x = e_1\ \texttt{in}\ e_1]$$

L'un des intérêts de l'évaluation paresseuse est qu'elle permet notamment de construire des "listes infinies".

Question 15 Dessiner l'ensemble des blocs alloués en mémoire à l'issue de l'évaluation de l'expression `let rec l = (:: 0) l in l`.

Question 16 Expliquer pourquoi l'évaluation de l'expression

```
let rec l = (:: 0) l in match l with [] -> 0 | x :: y -> x
```

termine.

Question 17 Que se passe-t-il si on tente d'évaluer l'expression `let rec x = x in x` ?

Évaluation paresseuse et effets de bord. On suppose qu'on ajoute une primitive `affiche` qui se comporte comme la fonction identité sur les entiers, mais a pour effet de bord d'afficher son argument. En particulier, cette primitive force l'évaluation de son argument.

Question 18 Indiquer ce qui est affiché pendant l'évaluation de l'expression suivante :

```
let x = affiche 1 in
let y = affiche 2 in
let z = affiche 3 in
((ifz z) ((+ y) z)) ((+ x) z)
```

Question 19 De manière générale, discuter la pertinence du mélange d'évaluation paresseuse et de traits impératifs (références, entrées-sorties, exceptions, etc.).

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`

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