Programmation Fonctionnelle Avancée

Contrôle des connaissances 2010/2011 – Rattrapage du 16 juin 2011
— Solution —
Master Informatique 2ème année, spécialité Programmation et Logiciels Sûrs
Institut Galilée – Université Paris 13

Luca Saiu

Documents de cours et de TP autorisés
Durée 3h

Le barème est donné à titre indicatif pour les quatre parties (total : **30 points**)

Première partie

Typage intuitif (5 points)

Q1. (5 points) Pour chaque expression OCaml suivante, dire si l'expression est bien typée et, le cas échéant, donner le type. En revanche, justifiez la réponse si vous considérez l'expression non typable.
Exemples : l'expression `42` a type `int` ; l'expression `1 + ()` n'est pas bien typée parce que l'opérande à droite du `+` n'est pas de type entier.

(a) ($\frac{1}{2}$ point) `[] :: []  : 'a list list`

(b) ($\frac{1}{2}$ point) `1 :: []  : int list`

(c) ($\frac{1}{2}$ point) `[] :: 1` — Erreur, l'object à droite de `::` n'est pas une liste

(d) ($\frac{1}{2}$ point) `[1] :: []` `: int list list`

(e) ($\frac{1}{2}$ point) `[] :: [1]` — Erreur : un object de type `'a list` ne peut pas être ajouté à une liste de type `int list` : la liste à droite de `::` est une liste d'entiers, pas une liste de listes

(f) ($\frac{1}{2}$ point) `fun x -> 1 : 'a -> int`

(g) ($\frac{1}{2}$ point) `fun x -> fun y -> x  : 'a -> 'b -> 'a`

(h) ($\frac{1}{2}$ point) `(fun x -> x)(let z = 1 in 2)  : int`

(i) ($\frac{1}{2}$ point) `fun x -> if x then x else x  : bool -> bool`

(j) ($\frac{1}{2}$ point) `if 1 then 2 else 3` — Erreur : la condition n'est pas booléenne

---

Deuxième partie

Interprètes et typage formel (10 points)

Q2. (4 points) Le type `expression` défini ci-dessous représente un arbre de syntaxe abstraite pour un langage d'expressions arithmétiques faites de nombres entiers, sommes et produits, mais sans variables.

```
type expression =
| Number of int
| Plus of expression * expression
| Times of expression * expression;;
```

Écrivez la définition d'une fonction `eval` de type `expression -> int` qui retourne la valeur d'une expression donnée.

```
let rec eval e =
  match e with
```

```
  | Number n -> n
  | Plus(e1, e2) ->  (eval e1) + (eval e2)
  | Times(e1, e2) -> (eval e1) * (eval e2);;
```

Q3. **(6 points)** Je n'ai jamais montré cette partie du langage au cours, mais OCaml comprend aussi un type tableau homogène prédéfini ; bien sûr le type est paramétrique, pareil au type liste : un tableau a type τ `array` lorsque ses éléments ont type τ.

On a aussi des fonction prédéfinies qui permettent de créer des tableaux, dont vous pouvez ignorez les détails. Ici nous sommes intéressés aux opérateurs d'accès aux tableaux, en lecture et en écriture. Formellement on peut décrire la syntaxe des opérateurs d'accès aux tableaux de la façon suivante :

$e ::= e\,.(e)$
$e ::= e\,.(e)$ `<-` e

Voilà une explication intuitive :

- Accès en lecture : en OCaml on écrit «$e_1\,.(e_2)$», si e_1 et e_2 sont des expressions OCaml, pour obtenir le contenu du tableau exprimé par e_1 à l'index exprimé par e_2 (en C ou Java, si e_1 et e_2 sont des expressions C ou Java, on écrit «$e_1[e_2]$»)
- Accès en écriture : si e_1, e_2 et e_3 sont des expressions OCaml, on écrit «$e_1\,.(e_2)$ `<-` e_3» pour représenter ce qui en C ou en Java serait écrit «$e_1[e_2]$ `=` e_3» (avec e_1, e_2 et e_3 des expressions C ou Java).

Donnez des règles de typage pour les opérateurs d'accès aux tableaux (je demande *des règles logiques, pas du code*) en justifiant votre réponse.

Les règles sont juste une traduction littérale de l'explication intuitive ci-dessus ; la seule chose à comprendre, très évidente en considérant les autres langages, est que l'index doit être entier.

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\,.(e_2) : \tau}$$

En français : si pour quelque type τ le type de e_1 est un tableau d'objets de type τ et si e_2 a type entier, alors $e_1\,.(e_2)$ a type τ.

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\,.(e_2) \texttt{ <- } e_3 : \texttt{unit}}$$

En français : si pour quelque type τ le type de e_1 est un tableau d'objets de type τ, si e_2 a type entier et si e_3 a type τ, alors l'affectation $e_1\,.(e_2)$ `<-` e_3 a type `unit`. En OCaml on utilise le type `unit` pour les résultats qui ne sont pas intéressants, par exemple les résultats des affectations.

L'environnement des types Γ est toujours le même car aucune variable n'est liée pas les opérateurs d'accès aux tableaux.

Troisième partie

Gestion automatique de la mémoire (4 points)

Q4. **(4 points)** Dans les fonctions OCaml suivantes, soulignez les expressions dont l'execution provoque l'allocation dynamique de la mémoire sur le *heap*, et (cerclez) les expressions dont l'execution provoque *toujours* la destruction de au moins un objet sur le *heap* :

```
let singleton x =
  [x];;

let rec append xs ys =
  match xs with
    | [] -> ys
    | first :: rest -> first :: (append rest ys);;

let rec reverse xs =
  match xs with
    | [] -> []
    | first :: rest -> append (reverse rest) (singleton first);;
```

Ici l'allocation sur le heap est provoquée par la construction des *cons*, y comprise l'expression `[x]` qui est juste une abréviation de `x :: []` ; on pourrait souligner aussi les appels de fonction qui allouent de la mémoire (tous les fonctions, dans ce cas). J'ai explicitement parlé des *expressions*, et on doit savoir les reconnaître : 2 points de moins par apport à cette question si vous avez souligné des motifs. Et non, le *pattern matching* ne provoque pas de l'allocation dynamique : rien n'est alloué, il s'agit juste d'analyser un objet déjà existant.

Si vous avez cerclé au moins une expression votre note par apport à cette question est zéro : il faut savoir que si on n'utilise que la gestion de la mémoire automatique, **on ne peut pas** contrôler où les objets sont détruits.

Quatrième partie

Modules et foncteurs (11 points)

Soient données les deux signatures suivantes :

```
module type ASignature = sig
  type t;;

  exception PredecessorOfZero;;
  (* Le nombre zéro, converti en t *)
  val zero : t;;

  (* is_zero retourne true si son parametre est zéro, sinon elle retourne false *)
  val is_zero : t -> bool;;
  (* predecessor retourne le prédecesseur de son paramètre, ou elle soulève
     PredecessorOfZero si son paramètre est zéro *)
  val predecessor : t -> t;;
  (* predecessor retourne le successeur de son paramètre *)
  val successor : t -> t;;
  (* to_int retourne son paramètre converti en entier OCaml *)
  val to_int : t -> int;;
end;;

module type BSignature = sig
  type t;;
  (* Le nombre un, converti en t *)
  val one : t;;

  (* sum retourne la somme de ses paramètres: *)
  val sum : t -> t -> t;;
  (* to_int retourne son paramètre converti en entier OCaml *)
  val to_int : t -> int;;
end;;
```

Q5. **(3 points)** Donnez la définition d'un module `A` qui respecte la signature `ASignature`, en respectant les commentaires aussi.

La solution la plus facile (mais pas la seule) est :

```
module A : ASignature = struct
  type t = int;;
  exception PredecessorOfZero;;

  let zero = 0;;
  let is_zero n = (n = 0);;
  let predecessor n = if n = 0 then raise PredecessorOfZero else (n - 1);;
  let successor n = n + 1;;
  let to_int n = n;;
end;;
```

Q6. **(5 points)** Donnez la définition d'un foncteur `MakeB` qui, donné un module avec signature `ASignature`, produit un module avec signature `BSignature`. Le type `t` défini dans le module retourné par le foncteur doit être le même type défini dans le module donné. Respectez les commentaires en `BSignature`.

```
module MakeB (SomeA : ASignature) : BSignature = struct
  type t = SomeA.t;;
  let one = SomeA.successor SomeA.zero;;
  let rec sum a b =
```

```
    if SomeA.is_zero a then
      b
    else
      SomeA.successor (sum (SomeA.predecessor a) b);;
  let to_int = SomeA.to_int;;
end;;
```

Q7. **(3 points)** Utilisez le module et le foncteur définis aux points précédents pour calculer $2+2$, en utilisant le type `t`, et affichiez le résultat.

```
module B = MakeB(A);;

let two = B.sum B.one B.one;;
let four = B.sum two two;;
Printf.printf "%i\n" (B.to_int four);;
```