Programmation Fonctionnelle Avancée

Contrôle des connaissances 2011/2012 – Partiel du 13 décembre 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 : **31 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) `match 42 with | _ -> true : bool`
(b) ($\frac{1}{2}$ point) `fun x -> match x with | [] -> true | _ -> false : 'a list -> bool`
(c) ($\frac{1}{2}$ point) `fun x -> x :: x` — Erreur, une liste ne peut jamais avoir le même type de ses éléments
(d) ($\frac{1}{2}$ point) `fun x y -> x :: y : 'a -> 'a list -> 'a list`
(e) ($\frac{1}{2}$ point) `let x = 1 in fun y -> x : 'a -> int`
(f) ($\frac{1}{2}$ point) `let x = 1 in let y = false in "abc" : string`
(g) ($\frac{1}{2}$ point) `(fun x -> x)(fun y -> y + 1) : int -> int`
(h) ($\frac{1}{2}$ point) `(fun x -> x)((fun y -> y + 1) 10) : int`
(i) ($\frac{1}{2}$ point) `let x = fun y -> y in (x true) : bool`
(j) ($\frac{1}{2}$ point) `fun x -> let y = x in y : 'a -> 'a`

Deuxième partie

Spécification formelle des langages (10 points)

Un *arbre binaire* est une structure de données définie par induction : un arbre peut soit être vide, soit contenir un sous-arbre gauche, un élément racine et un sous-arbre droit. Les arbres sont *homogènes* : dans chaque arbre binaire les données dans tous les racines doivent avoir le même type (mais pas nécessairement entier).

Voilà deux exemples — le deuxième est l'arbre vide. Remarquez que les feuilles ne contiennent jamais des données.

On veut ajouter les arbres binaires à *trivialML*.
Clarification : aucune question de cette deuxième partie ne demande d'écrire du code OCaml.

Q2 : syntaxe. (2 points) Ajoutez à la grammaire de *trivialML* le support pour des expressions de type arbre (écrivez juste las partie que vous ajoutez). Inventez une syntaxe raisonnable ; si vous voulez vous pouvez utiliser la virgule «`,`» ou le point «`.`», les accolades «`{`» et «`}`» ou les crochets «`[`» et «`]`», qui ne sont encore utilisées en *trivialML*.

Je vous rappelle que un arbre peut être soit vide, soit non vide.

Une solution possible :

τ ::=
  τ `arbre`

e ::=
  `{ }`
| `{`e `.` e `.` e`}`

Autre solution : ajouter le cas «arbre vide» aux constantes littérales c.

En tout cas, zéro points si dans le cas récursif vous n'avez pas écrit «e» trois fois : il faut comprendre que chaque sous-expression peut être un expression arbitraire.

Q3 : valeurs. (1 point) Il faut modifier la définition des valeurs *trivialML* de façon que les arbres binaires soient supportés aussi. Écrivez vos modifications (en français ou en anglais) du texte à la section 2.1.2 de « *The trivialML language : formal syntax, semantics and typing*».

Une solution possible : les valeurs peuvent être aussi des arbres binaires ; nous représentons la valeur «arbre vide» par «$\mathcal{V}$», et la valeur «arbre non vide avec sous arbre gauche t_1, racine v et sour-arbre droit t_2» par «$\mathcal{A}(t_1, v, t_2)$».

Q4 : sémantique. (2 points) Il faut modifier la sémantique de *trivialML* de façon que les arbres binaires soient supportés aussi. Écrivez vos modifications de la définition de $E[\![\_]\!]\_\ \_$ (voire la section 2.2 de « *The trivialML language : formal syntax, semantics and typing*»).

On ajoute :

$E[\![\texttt{\{ \}}]\!]\rho\Gamma = \mathcal{V}$
$E[\![\texttt{\{}e_1\ \texttt{.}\ e_2\ \texttt{.}\ e_3\texttt{\}}]\!]\rho\Gamma = \mathcal{A}(E[\![e_1]\!]\rho\Gamma,\ E[\![e_2]\!]\rho\Gamma,\ E[\![e_3]\!]\rho\Gamma)$

Bien sûr, une solution qui utilise correctement des blocs *let..in* est parfaitement acceptable aussi.

Q5 : règles de typage. (3 points) Donnez des règles de typage pour les arbres binaires (voire la section 3.1 de « *The trivialML language : formal syntax, semantics and typing*»).

$$[\mathcal{V}]\ \frac{}{\Sigma \vdash \texttt{\{ \}} : \tau\ \texttt{arbre}}$$

$$[\mathcal{A}]\ \frac{\Sigma \vdash e_1 : \tau\ \texttt{arbre} \qquad \Sigma \vdash e_2 : \tau \qquad \Sigma \vdash e_3 : \tau\ \texttt{arbre}}{\Sigma \vdash \texttt{\{}e_1\ \texttt{.}\ e_2\ \texttt{.}\ e_3\texttt{\}} : \tau\ \texttt{arbre}}$$

Q6. (2 points) Est-ce que les modifications du langage demandées par les questions **Q2**, **Q3**, **Q4** et **Q5** suffisent à écrire des programmes *trivialML* utiles qui travaillent sur des arbres ? Par exemple, peut-on écrire une fonction *trivialML* qui calcule l'hauteur d'un arbre donné ? Justifiez votre réponse.

La réponse est *non*. Nous pouvons créer des arbres, mais nous n'avons aucune opération pour travailler sur des arbres existants : par exemple pour calculer l'hauteur d'un arbre donné il faudrait avoir au moins un opérateur pour déterminer si l'arbre est vide, et des sélecteurs pour extraire les sous-arbres d'un arbre donné. En général (pas pour le calcul de l'hauteur) on a aussi besoin d'un opérateur pour extraire la racine d'un arbre donné.

Troisième partie

Typage formel avancé (6 points)

On suppose avoir ajouté à *trivialML* les *fonctions récursives localement nommées*.

e ::=
  `fonction_récursive` x_1`:`τ_1 `.` x_2`:`τ_2 `->` e

Une fonction récursive localement nommée est une fonction dont le nom (x_1 ci-dessus) est visible dans son corp, et *que* dans son corp. Puisque en *trivialML* nous n'avons pas d'inférence de type, les déclarations du type de la fonction (τ_1 ci-dessus) *et* du type de son paramètre (τ_2 ci-dessus) sont obligatoires.
Encore une fois, remarquez que x_1 est le nom de la fonction et x_2 est le nom de son paramètre ; τ_1 est le type de la fonction et τ_2 est le type de son paramètre.

Exemple (une expression) :

```
((fonction_récursive f : entier -> entier . n : entier = si n = 0 alors 1 sinon n * (f (n - 1))) 10)
```

Remarquez que dans l'exemple on a appliqué une fonction récursive sans la définir globalement, ce qui est impossible en *trivialML* sans cette extension.

Q7. (6 points) Écrivez la règle de typage pour les fonctions récursives localement nommées.

$$[\texttt{réc}]\ \frac{\Sigma[x_1 : \tau_1 \texttt{ -> } \tau_2,\quad x_2 : \tau_1] \vdash e : \tau_2}{\Sigma \vdash (\texttt{fonction\_récursive}\ x_1 : \tau_1 \texttt{ -> } \tau_2\ .\ x_2 : \tau_1 \texttt{ -> } e) : \tau_1 \texttt{ -> } \tau_2}$$

Quatrième partie
Gestion automatique de la mémoire (5 points)

Q8. (5 points) Donnez un exemple (complet, sans «...») de code OCaml où du *garbage sémantique* qui n'est pas du *garbage syntaxique* est retenu par le *garbage collector*, en justifiant votre réponse. Votre exemple ne doit pas nécessairement être réaliste ou utile en pratique.

Vous pouvez penser que le compilateur soit «bête», c'est-à-dire qu'il n'élimine ni déplace jamais les allocations.

```
let rec make_list n =
  if n = 0 then
    []
  else
    n :: (make_list (n - 1));;
let useless = make_list 100000 in
while true do
  ();
done;;
```

La liste liée à la variable `useless` n'est jamais utilisée après sa création, donc elle dévient immédiatement *garbage sémantique* ; mais en tant que variable globale elle reste toujours accessible, donc elle ne devient jamais *garbage syntaxique*.
Solution alternative : on peut utiliser un bloc `let..in` au lieu d'une définition globale pour lier `useless`, avec la même boucle `while` dans le corps.

Cinquième partie
Modules et foncteurs (5 points)

La signature `ListSignature` ci-dessous spécifie une interface de module qui implémente des listes homogènes, similaires aux listes prédéfinies OCaml :

```
module type ListSignature = sig
  type 'a t;; (* notre liste d'éléments de type 'a *)

  val empty : 'a t;; (* la liste vide *)
  val is_empty : 'a t -> bool;; (* retourne true si et seulement si
                                   le paramètre est la liste vide *)
  val cons : 'a -> 'a t -> 'a t;; (* :: *)
  val head : 'a t -> 'a;; (* retourne la tête de la liste donnée, ou
                             échoue si la liste est vide *)
  val tail : 'a t -> 'a t;; (* retourne la queue de la liste donnée, ou
                               échoue si la liste est vide *)
end;;
```

La signature `ExtendedListSignature` contient tout ce qui est déclaré en `ListSignature`, et trois nouvelles fonctions de plus :

```
module type ExtendedListSignature = sig
  (* Cette première partie et pareille à ListSignature *)
```

```
  type 'a t;;
  val empty : 'a t;;
  val is_empty : 'a t -> bool;;
  val cons : 'a -> 'a t -> 'a t;;
  val head : 'a t -> 'a;;
  val tail : 'a t -> 'a t;;

  (* Les trois nouvelles fonctions: *)
  val length : 'a t -> int;; (* taille d'une liste *)
  val to_predefined : 'a t -> 'a list;; (* conversion de notre liste à
                                           une liste standard OCaml *)
  val of_predefined : 'a list -> 'a t;; (* conversion de une liste
                                           standard OCaml à notre liste *)
end;;
```

Q8. **(5 points)** Implémentez en OCaml un foncteur qui accepte en entrée un module avec signature `ListSignature` et produit en sortie un module avec signature `ExtendedListSignature`. Respectez les commentaires.

```
module MakeExtendedList(TheList : ListSignature) : ExtendedListSignature = struct
  type 'a t = 'a TheList.t;;
  let empty = TheList.empty;;
  let is_empty = TheList.is_empty;;
  let cons = TheList.cons;;
  let head = TheList.head;;
  let tail = TheList.tail;;
  let rec length xs =
    if is_empty xs then
      0
    else
      1 + length (tail xs);;
  let rec of_predefined xs =
    match xs with
      | [] -> empty
      | first :: rest -> cons first (of_predefined rest);;
  let rec to_predefined xs =
    if is_empty xs then
      []
    else
      (head xs) :: (to_predefined (tail xs));;
end;;
```