Programmation Fonctionnelle Avancée

Contrôle des connaissances 2010/2011
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 : 26 points)

Première partie

Compréhension de code OCaml et typage (8 points)

Q1. Pour chaque expression suivante, dire si l'expression est correcte et, le cas échéant, donner le type. En revanche, justifiez la réponse si vous considérez l'expression non typable.

(a) `1 :: 2`

(b) `[true] :: []`

(c) `(fun q -> []) (fun y -> y)`

(d) `fun x -> 3`

(e) `fun x -> fun y -> x`

(f) `fun x -> (x, x)`

(g) `let x = 1 in let y = "a" in [x, y] (* Oui, virgule, pas ‘‘;’’ *)`

(h) `let x = 1 in let x = "a" in x (* Oui, c'est le même nom x *)`

(i) `(1, "a")`

(j) `[1, "a"] (* Oui, virgule, pas ‘‘;’’ *)`

(k) `let x = 1 in if x < x then x else ()`

Q2. Écrire une expression dont le type soit :

(a) `bool -> int`

(b) `('a * 'b) -> 'a`

(c) `'a -> ('a * int * ('a list))`

Q3. Voilà une définition possible de la fonction `map2` :

```
let rec map2 f a b =
  match a, b with
  | [], [] ->
      []
  | (first_a :: rest_a), (first_b :: rest_b) ->
      (f first_a first_b) :: (map2 f rest_a rest_b);;
```

L'interprète répond à la définition avec le type inféré...

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
```

...mais il montre un message de *warning* aussi.

(*a*) Pourquoi le message de warning est là ? Expliquez le problème.

(*b*) Donnez un exemple d'application de `map2` démontrant le problème prévu dans le message de warning.

(*c*) Donnez une définition alternative de la fonctionne `map2` qui évite le message de warning. Vous allez avoir besoin d'une exception.

(*d*) `let foo x y = map2 (fun a b -> (a +. b) /. 2.0) x y;;`
Qu'est-ce que calcule la fonction `foo` ? Quel est son type ?

Deuxième partie
Typage formel (5 points)

Q4. Supposons d'avoir étendu *TrivialML* en supportant des listes homogènes, (c'est-à-dire avec tous les éléments du même type) ; nous ajoutons aussi une boucle `foreach` pour iterer sur chaque élément d'une liste donnée. Des boucles similaires à `foreach` existent dans beaucoup de langages (Java, Python, JavaScript, Python, ...).

Un exemple avec `foreach`, affichant les nombres `10`, puis `2` et enfin `4` :

```
foreach x : int in [10; 2; 4] do
  print_int x;;
- : unit = ()
```

L'expression suivante affiche la chaîne de caractères `"a"`, et puis la chaîne de caractères `"b"` :

```
let a = "a" in
foreach x : string in a :: ["b"] do
  print_string x;;
- : unit = ()
```

TrivialML est un langage typé statiquement, avec des règles de typage comme la suivante (pour le bloc `let`) :

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x : \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2}$$

La règle logique de `let` correspond à ce code de *type-checking* OCaml :

```
let rec expression_type expression rho gamma =
  match expression with
    ...
    | ELet(x, x_type, e1, e2) ->
      let e1_type = expression_type e1 rho gamma in
      if x_type = e1_type then
        let e2_type =
          expression_type e2 (Environment.bind x x_type rho) gamma in
        e2_type
      else
        failwith "wrong type declaration in let"
    ...
```

(*a*) Donnez une règle de typage pour `foreach` :

$$\frac{? \qquad ?}{\Gamma \vdash \texttt{foreach}\ x : \tau_1\ \texttt{in}\ e_1\ \texttt{do}\ e_2 : ?}$$

(*b*) Écrivez le code de *type-checkng* pour `foreach`. Signalez les erreurs de type en utilisant des "messages d'erreur" raisonnables, comme dans l'exemple ci-dessus (en français ou en anglais).

Troisième partie
Gestion automatique de la mémoire (5 points)

Q5. Supposons un langage fonctionnel similaire à OCaml, avec gestion automatique de la mémoire. La fonction `make_big_list : unit -> int list` alloue sur le *heap* une nouvelle, grande liste (par exemple, avec dix millions d'éléments) ; la fonction `make_small_list : unit -> int list` alloue sur le *heap* une petite liste (par exemple, avec dix éléments). Soit `big_int` un nombre entier très grand, par exemple `100000000`.

Supposons un compilateur "stupide", qui n'exécute aucune optimisation.

(*a*) On exécute le programme suivant :

```
let x = make_big_list () in
  for i = 1 to big_int do
    let _ = make_small_list () in
    ();
  done;
  List.length x;;
```

Pour ce programme quel système de gestion automatique de la mémoire convient utiliser ? Un conteur de références ou un *tracing garbage collector* ? Pourquoi ? (Si vous connaissez les ramasses-miettes à générations, que je n'ai pas expliqué au cours, considérez juste un système avec une seule génération)

(*b*) Le programme suivant contient une boucle infinie :

```
let x = make_big_list () in
  while true do
    let _ = make_small_list () in
    ();
  done;
  List.length x;;
```

Est-ce que la grande liste liée à la variable `x` est jamais détruite ? Pourquoi ?

Quatrième partie

Foncteurs (8 points)

Q6. La signature `NatPlusInterface` reprisent l'interface d'un module pour travailler avec un type abstrait $\mathbb{N}^+ = \{1, 2, 3, ...\}$:

```
module type NatPlusInterface = sig
  type t;;

  exception PredecessorOfOne;;

  val one : t;;

  val is_one : t -> bool;;
  val successor : t -> t;;   (* this never fails *)
  val predecessor : t -> t;; (* this raises exception on one *)
end;;
```

Remarquez que $0 \notin \mathbb{N}^+$.

La signature `IntInterface` reprisent l'interface d'un module pour travailler avec les entiers $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$; les entiers aussi sont implémentés comme un type abstrait :

```
module type IntInterface = sig
  type t;;
  type sign = ZeroSign | PositiveSign | NegativeSign;;

  val zero : t;;

  val sign : t -> sign;;

  val successor : t -> t;;   (* this never fails *)
  val predecessor : t -> t;; (* this never fails *)
end;;
```

(*a*) Écrivez la définition d'un foncteur pour construire un module avec la signature `IntInterface` à partir d'un module `NatPlus` avec la signature `NatPlusInterface`. Le type `t` doit avoir la définition :

```
type t =
  | Zero
  | Positive of NatPlus.t
  | Negative of NatPlus.t;;
```